

# Linear Recursion\*

Sandra Alves<sup>1</sup>, Maribel Fernández<sup>2</sup>, Mário Florido<sup>1</sup>, and Ian Mackie<sup>3</sup>

<sup>1</sup> University of Porto, Faculty of Science & LIACC,  
R. do Campo Alegre 1021/55, 4169-007, Porto, Portugal

<sup>2</sup> King's College London, Department of Informatics  
Strand, London WC2R 2LS, U.K.

<sup>3</sup> LIX, CNRS UMR 7161, École Polytechnique  
91128 Palaiseau Cedex, France

## Abstract

We show that the full PCF language can be encoded in  $\mathcal{L}_{\text{rec}}$ , a linear  $\lambda$ -calculus extended with numbers, pairs and an unbounded recursor that preserves the syntactic linearity of the calculus. Thus,  $\mathcal{L}_{\text{rec}}$  is a Turing complete extension of the linear  $\lambda$ -calculus. We discuss evaluation strategies and implementation techniques for  $\mathcal{L}_{\text{rec}}$ , exploiting the linearity of the system.

## 1 Introduction

Knowing that an argument to a function is used exactly once—i.e., linearly—is a property that a compiler can make use of to optimise code. It is related to several program analyses, for instance, strictness analysis, pointer analysis, resource analysis (see, e.g., [14, 15, 22, 23, 58, 56, 57, 21, 37]); computing these analyses give approximations. Linear functions are also naturally occurring in hardware compilation [28]. Circuits are static (i.e., they cannot be copied at run-time), so linear computations are more naturally compiled into hardware.

Linearity has also applications in other domains. For instance, in the area of quantum computation one of the most important results is the no-cloning theorem, stating that qbits cannot be duplicated. Again, a linear calculus captures this [55]. In concurrent calculi, like the  $\pi$ -calculus [51], a key aspect is the notion of name, and the dual role that names play as communication channels and variables. The linear  $\pi$ -calculus [46] has linear (use-once) channels. This has clear gains in efficiency and on program analysis avoiding several problems of channel sharing. Also, inspired by the works by Kobayashi, Pierce and Turner [46] and the works by Honda [39] on session types, several type systems for the  $\pi$ -calculus rely directly on linearity to deal with resources, non-interference and effects [34, 45, 59].

In this paper we focus on functional computations. We aim at obtaining a linear, universal model of computation that can serve as a basis for the design of programming languages. Our approach is to begin with a linear calculus, and build non-linearity in a controlled way.

Several extensions of the linear  $\lambda$ -calculus, based on bounded iteration, proved to capture interesting classes of programs (see, e.g., [29, 33, 7, 8, 9, 36, 48, 54]). In particular, previous results have shown that primitive recursive functions (PR) can be encoded in linear versions of Gödel's System  $\mathcal{T}$  [18]. Furthermore, System  $\mathcal{L}$  [4], a linear  $\lambda$ -calculus extended with pairs, numbers and a bounded iterator, with a closed reduction strategy [24], has all the computation power of System  $\mathcal{T}$ . The latter result shows some redundancy regarding duplication in System  $\mathcal{T}$ ,

---

\*This paper is an extended version, with detailed proofs, of [5].

which can be achieved through iteration or through non-linear occurrences of the bound variable in the body of a function.

Following this work, the question that arises is, what is the minimal extension of the linear  $\lambda$ -calculus that yields a Turing complete system, compatible with the notion of linear function?

From the perspective of recursion theory, Turing completeness can be achieved by adding a minimisation operator working on a first-order linear system (using a set of linear initial functions and a linear primitive recursion scheme) [3]. A similar result is shown in this paper for the linear  $\lambda$ -calculus. More precisely, we show that an extension of System  $\mathcal{L}$  with a minimiser is Turing complete. However, it relies on both iteration and minimisation.

In the context of the simply typed  $\lambda$ -calculus, there is an alternative way to obtain a Turing complete system, by adding a fixpoint operator (as it is done in PCF [53]). This approach has been used to extend linear functional calculi (see, for instance, [49, 13, 52, 16, 19]), however, it relies on the existence of a non-linear conditional which throws away a possibly infinite computation in one of the branches. Instead, in this paper, we introduce recursion in the linear  $\lambda$ -calculus through the use of an unbounded recursor with a built-in test on pairs, which allows the encoding of both finite iteration and minimisation. More precisely, we define  $\mathcal{L}_{\text{rec}}$ , a linear  $\lambda$ -calculus extended with numbers, pairs and a linear unbounded recursor, with a closed-reduction strategy. We show that  $\mathcal{L}_{\text{rec}}$  is Turing complete and can be easily implemented: we give an abstract machine whose configurations consist simply of a term and a stack of terms. As an application, we give a compilation of PCF into  $\mathcal{L}_{\text{rec}}$ .

Several abstract machines for linear calculi are available in the literature (see for instance [50, 57, 47]). The novelty of our approach is that we implement a calculus that is syntactically linear (in the sense that each variable is linear in  $\mathcal{L}_{\text{rec}}$  terms) and therefore there is no need to include in the abstract machine an environment (or store in the terminology of [57]) to store bindings for variables.

Summarising, the main contributions of this paper are the following:

- We define  $\mathcal{L}_{\text{rec}}$ , a linear  $\lambda$ -calculus extended with numbers, pairs and an unbounded recursor, with a closed-reduction strategy. We show some properties regarding reduction (such as subject-reduction and confluence), and prove that  $\mathcal{L}_{\text{rec}}$  is Turing complete by encoding the set of partial recursive functions in  $\mathcal{L}_{\text{rec}}$ .
- We give call-by-name and call-by-value evaluation strategies, and define a simple abstract machine for  $\mathcal{L}_{\text{rec}}$ , exploiting its linearity.
- We study the interplay between linearity and recursion based on fixpoint combinators, and define an encoding of PCF in  $\mathcal{L}_{\text{rec}}$ .
- By combining the two previous points, we obtain a new implementation of PCF via a simple stack-based abstract machine.

**Related notions of linearity** Three notions of linearity have been defined for functional calculi in the literature: syntactical, operational and denotational. Operational linearity means that arguments of functions cannot be duplicated or erased during evaluation (cf. *weak linear terms* [6] and *simple terms* [43]). Denotational linearity is achieved when only linear functions can be defined in the language [27, 52]. The language defined in [52] is a linear version of PCF in a denotational sense: it has a linear model (linear coherence spaces) but its terms can contain more than one occurrence of the same variable. Finally, syntactical linearity, requires a linear use of variables in terms [40].

Operational linearity has great impact when the control of copying and deleting is important, as it can be used to efficiently implement garbage collection for instance. Note however that checking if a term is operationally linear relies on the evaluation of the term. For the linear  $\lambda$ -calculus, syntactical linearity, which can be statically checked, implies operational linearity. For  $\mathcal{L}_{\text{rec}}$ , which combines syntactical linearity with closed reduction, the fragment without recursion is operationally linear; erasing and duplication can only be done by the recursor (in linear logic [30])

this is done by the use of exponentials, and in other linear calculi [1, 49, 38, 57] by explicit syntactical constructs). Moreover, only closed terms can be erased or duplicated in  $\mathcal{L}_{\text{rec}}$ , thanks to the use of a closed reduction strategy.

The rest of this paper is structured as follows. In the next section we introduce basic concepts and notations that will be needed in the rest of the paper. We give the syntax and type system of System  $\mathcal{L}_{\text{rec}}$  together with some standard properties in Section 3. In Section 4 we study evaluation strategies and abstract machines. In Section 5 we look at the relation with PCF. Finally, we conclude in Section 6.

## 2 Background

We assume the reader is familiar with the  $\lambda$ -calculus [10]. In this section we recall the definition of System  $\mathcal{L}$  [4], a linear version of Gödel's System  $\mathcal{T}$  (for details on the latter see [32]).

The terms of System  $\mathcal{L}$  are obtained by extending the terms of the linear  $\lambda$ -calculus [1] with numbers, pairs, and an iterator. Linear  $\lambda$ -terms  $t, u, \dots$  are inductively defined by:  $x \in \Lambda$ ,  $\lambda x.t \in \Lambda$  if  $x \in \text{fv}(t)$ , and  $tu \in \Lambda$  if  $\text{fv}(t) \cap \text{fv}(u) = \emptyset$ . Note that  $x$  is used at least once in the body of the abstraction, and the condition on the application ensures that all variables are used at most once. Thus these conditions ensure syntactic linearity (variables occur exactly once). In System  $\mathcal{L}$  we also have numbers generated by 0 and S, with an iterator:

$$\text{iter } t \ u \ v \text{ if } \text{fv}(t) \cap \text{fv}(u) = \text{fv}(u) \cap \text{fv}(v) = \text{fv}(v) \cap \text{fv}(t) = \emptyset$$

and pairs:

$$\begin{aligned} \langle t, u \rangle & \text{ if } \text{fv}(t) \cap \text{fv}(u) = \emptyset \\ \text{let } \langle x, y \rangle = t \text{ in } u & \text{ if } x, y \in \text{fv}(u) \text{ and} \\ & \text{fv}(t) \cap (\text{fv}(u) - \{x, y\}) = \emptyset \end{aligned}$$

Since  $\lambda$  and **let** are binders, terms are defined modulo  $\alpha$ -equivalence as usual.

Note that when projecting from a pair, we use both projections. A simple example is the function that swaps the components of a pair:  $\lambda x.\text{let } \langle y, z \rangle = x \text{ in } \langle z, y \rangle$ . In examples below we use tuples of any size, built from pairs. For example,  $\langle x_1, x_2, x_3 \rangle = \langle x_1, \langle x_2, x_3 \rangle \rangle$  and  $\text{let } \langle x_1, x_2, x_3 \rangle = u \text{ in } t$  represents the term  $\text{let } \langle x_1, y \rangle = u \text{ in let } \langle x_2, x_3 \rangle = y \text{ in } t$ .

System  $\mathcal{L}$  uses a closed reduction strategy (first defined by Girard [31] for cut elimination in linear logic, and adapted to the  $\lambda$ -calculus in [24]). This strategy for cut elimination is simple and exceptionally efficient in terms of the number of cut elimination steps. In the  $\lambda$ -calculus, it avoids  $\alpha$ -conversion while allowing reductions inside abstractions (in contrast with standard weak strategies), thus achieving more sharing of computation.

The reduction rules for System  $\mathcal{L}$  are given in Table 1. Substitution is a meta-operation defined as usual, and reductions can take place in any context.

Name	Reduction	Condition
<i>Beta</i>	$(\lambda x.t)v \rightarrow t[v/x]$	$\text{fv}(v) = \emptyset$
<i>Let</i>	$\text{let } \langle x, y \rangle = t, u \text{ in } v \rightarrow (v[t/x])[u/y]$	$\text{fv}(t) = \text{fv}(u) = \emptyset$
<i>Iter</i>	$\text{iter } (S \ t) \ u \ v \rightarrow v(\text{iter } t \ u \ v)$	$\text{fv}(v) = \emptyset$
<i>Iter</i>	$\text{iter } 0 \ u \ v \rightarrow u$	$\text{fv}(v) = \emptyset$

Table 1: Closed reduction in System  $\mathcal{L}$

Note that all the substitutions created during reduction (rules *Beta* and *Let*) are closed (thus, there is no need to perform  $\alpha$ -conversions during reduction), and the *Iter* rules are only triggered when the function  $v$  is closed. Thanks to the use of a closed reduction strategy, iterators on *open* linear functions are accepted in System  $\mathcal{L}$  (since these terms are syntactically linear), and reduction preserves linearity. Normal forms are not the same as in the  $\lambda$ -calculus (for example,  $\lambda x.(\lambda y.y)x$  is a normal form), but closed reduction is still adequate for the evaluation of closed

Construction	Variable Constraint	Free Variables (fv)
0	—	$\emptyset$
S $t$	—	$\text{fv}(t)$
rec $t_1 t_2 t_3 t_4$	$\text{fv}(t_i) \cap \text{fv}(t_j) = \emptyset$ , for $i \neq j$	$\cup \text{fv}(t_i)$
$x$	—	$\{x\}$
$tu$	$\text{fv}(t) \cap \text{fv}(u) = \emptyset$	$\text{fv}(t) \cup \text{fv}(u)$
$\lambda x.t$	$x \in \text{fv}(t)$	$\text{fv}(t) \setminus \{x\}$
$\langle t, u \rangle$	$\text{fv}(t) \cap \text{fv}(u) = \emptyset$	$\text{fv}(t) \cup \text{fv}(u)$
let $\langle x, y \rangle = t$ in $u$	$x, y \in \text{fv}(u), \text{fv}(t) \cap \text{fv}(u) = \emptyset$	$\text{fv}(t) \cup (\text{fv}(u) \setminus \{x, y\})$

Table 2: Terms in System  $\mathcal{L}_{\text{rec}}$

Name	Reduction	Condition
<i>Rec</i> <b>rec</b> $\langle 0, t' \rangle u v w$	$\rightarrow u$	$\text{fv}(t'vw) = \emptyset$
<i>Rec</i> <b>rec</b> $\langle S t, t' \rangle u v w$	$\rightarrow v(\text{rec } (w \langle t, t' \rangle) u v w)$	$\text{fv}(vw) = \emptyset$

Table 3: Closed reduction for recursion

terms (if a term has a weak head normal form, it will be reached [4]). Closed reduction can also be used to evaluate open terms, using the “normalisation by evaluation” technique [12] as shown in [24, 25] (in the latter director strings are used to implement closedness tests as local checks on terms).

Although linear, some terms are not strongly normalising. For instance,  $\Delta\Delta$  where  $\Delta = \lambda x.\text{iter } S^2 0 (\lambda xy.xy) (\lambda y.yx)$  reduces to itself. However, the linear type system defined in [4] ensures strong normalisation. System  $\mathcal{L}$  has all the power of System  $\mathcal{T}$ ; we refer to [4] for more details and examples.

### 3 System $\mathcal{L}_{\text{rec}}$ : syntax and properties

System  $\mathcal{L}$  is not Turing complete. In this section we define  $\mathcal{L}_{\text{rec}}$ , an extension of the linear  $\lambda$ -calculus [1] with numbers, pairs, and a typed unbounded recursor with a closed reduction strategy that preserves syntactic linearity. We prove that this system is Turing complete.

The syntax of System  $\mathcal{L}_{\text{rec}}$  is similar to that of System  $\mathcal{L}$ , except that instead of a bounded iterator we have a recursor working on pairs of natural numbers. Table 2 summarises the syntax of terms in  $\mathcal{L}_{\text{rec}}$ . We assume Barendregt’s convention regarding names of free and bound variables in terms.

The reduction rules for  $\mathcal{L}_{\text{rec}}$  are *Beta* and *Let*, given in Table 1, together with two rules for the recursor shown in Table 3.

Note that the *Rec* rules are only triggered when the closedness conditions hold, thus linearity is preserved by reduction. The conditions on *Beta* and *Let* are orthogonal to the linearity issues (as explained in the previous section, they simply produce a more efficient strategy of reduction) and do not affect the technical results of the paper (even with these conditions, the system will be shown to be Turing complete).

The *Rec* rules pattern-match on a pair of numbers (the usual bounded recursor works on a single number). This is because we are representing both bounded and unbounded recursion with the same operator (as the examples below illustrate). An alternative would be to have an extra parameter of type  $\mathbb{N}$  in the recursor.

### 3.1 Examples

**Bounded iteration** Using the recursor we can encode System  $\mathcal{L}$ 's iterator. Let  $I$  be the identity function  $\lambda x.x$ . We define “iter” in System  $\mathcal{L}_{\text{rec}}$  as follows:

$$\text{“iter” } t \ u \ v \stackrel{\text{def}}{=} \text{rec } \langle t, 0 \rangle \ u \ v \ I$$

We will show later that this term has the same behaviour as System  $\mathcal{L}$ 's iterator.

**Projections and duplication of natural numbers** We can define projections for pairs  $\langle a, b \rangle$  of natural numbers, by using them in a recursor.

$$\begin{aligned} pr_1 &= \lambda x. \text{let } \langle a, b \rangle = x \text{ in rec } \langle b, 0 \rangle \ a \ I \ I \\ pr_2 &= \lambda x. \text{let } \langle a, b \rangle = x \text{ in rec } \langle a, 0 \rangle \ b \ I \ I \end{aligned}$$

The following function  $C$  can be used to copy numbers:

$$C = \lambda x. \text{rec } \langle x, 0 \rangle \ \langle 0, 0 \rangle \ (\lambda x. \text{let } x = \langle a, b \rangle \text{ in } \langle Sa, Sb \rangle) \ I$$

Other mechanisms to erase and copy numbers in  $\mathcal{L}_{\text{rec}}$  will be shown later.

**Arithmetic functions** We can now define some arithmetic functions that we will use in the paper.

- $\text{add} = \lambda mn. \text{rec } \langle m, 0 \rangle \ n \ (\lambda x. Sx) \ I;$
- $\text{mult} = \lambda mn. \text{iter } m \ 0 \ (\text{add } n);$
- $\text{pred} = \lambda n. pr_1(\text{rec } \langle n, 0 \rangle \ \langle 0, 0 \rangle \ F \ I)$   
   where  $F = \lambda x. \text{let } \langle t, u \rangle = C(pr_2 \ x) \text{ in } \langle t, S \ u \rangle;$
- $\text{iszero} = \lambda n. pr_1(\text{rec } \langle n, 0 \rangle \ \langle 0, S \ 0 \rangle \ (\lambda x. C(pr_2 \ x)) \ I).$

The correctness of these encodings can be easily proved by induction.

**Minimisation** We can also encode the minimisation operator  $\mu_f$  used to define partial recursive functions. Recall that if  $f : \mathbb{N} \rightarrow \mathbb{N}$  is a total function on natural numbers,  $\mu_f = \min\{x \in \mathbb{N} \mid f(x) = 0\}$ .

Let  $\bar{f}$  be a closed  $\lambda$ -term in  $\mathcal{L}_{\text{rec}}$  representing a total function  $f$  on natural numbers. The encoding of  $\mu_f$  is

$$M = \text{rec } \langle \bar{f}0, 0 \rangle \ 0 \ (\lambda x. S(x)) \ F$$

where  $F = \lambda x. \text{let } \langle y, z \rangle = C(pr_2 x) \text{ in } \langle \bar{f}(Sy), Sz \rangle$ . We prove the correctness of this encoding below.

### 3.2 Types for System $\mathcal{L}_{\text{rec}}$

We consider *linear types* generated by the grammar:

$$A, B ::= \mathbf{N} \mid A \multimap B \mid A \otimes B$$

where  $\mathbf{N}$  is the type of numbers. A type environment  $\Gamma$  is a list of type assumptions of the form  $x : A$  where  $x$  is a variable and  $A$  a type, and each variable occurs at most once in  $\Gamma$ . We write  $\text{dom}(\Gamma)$  to denote the set of variables that occur in  $\Gamma$ .

We write  $\Gamma \vdash_{\mathcal{L}} t : A$  if the term  $t$  can be assigned the type  $A$  in the environment  $\Gamma$  using the typing rules in Table 4. Note that the only structural rule is Exchange, we do not have Weakening and Contraction rules: we are in a linear system. For the same reason, the logical rules split the context between the premises (i.e., the variable conditions in Table 2 are enforced by the typing rules).

All the terms given in the examples above can be typed.

**Axiom and Structural Rule:**

$$\frac{}{x : A \vdash_{\mathcal{L}} x : A} \text{ (Axiom)} \quad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{L}} t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{L}} t : C} \text{ (Exchange)}$$

**Logical Rules:**

$$\frac{\Gamma, x : A \vdash_{\mathcal{L}} t : B}{\Gamma \vdash_{\mathcal{L}} \lambda x. t : A \multimap B} \text{ } (\multimap \text{Intro}) \quad \frac{\Gamma \vdash_{\mathcal{L}} t : A \multimap B \quad \Delta \vdash_{\mathcal{L}} u : A}{\Gamma, \Delta \vdash_{\mathcal{L}} tu : B} \text{ } (\multimap \text{Elim})$$

$$\frac{\Gamma \vdash_{\mathcal{L}} t : A \quad \Delta \vdash_{\mathcal{L}} u : B}{\Gamma, \Delta \vdash_{\mathcal{L}} \langle t, u \rangle : A \otimes B} \text{ } (\otimes \text{Intro}) \quad \frac{\Gamma \vdash_{\mathcal{L}} t : A \otimes B \quad \Delta, x : A, y : B \vdash_{\mathcal{L}} u : C}{\Gamma, \Delta \vdash_{\mathcal{L}} \text{let } \langle x, y \rangle = t \text{ in } u : C} \text{ } (\otimes \text{Elim})$$

**Numbers**

$$\frac{}{\vdash_{\mathcal{L}} 0 : \mathbb{N}} \text{ (Zero)} \quad \frac{\Gamma \vdash_{\mathcal{L}} n : \mathbb{N}}{\Gamma \vdash_{\mathcal{L}} S n : \mathbb{N}} \text{ (Succ)}$$

$$\frac{\Gamma \vdash_{\mathcal{L}} t : \mathbb{N} \otimes \mathbb{N} \quad \Theta \vdash_{\mathcal{L}} u : A \quad \Delta \vdash_{\mathcal{L}} v : A \multimap A \quad \Sigma \vdash_{\mathcal{L}} w : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}}{\Gamma, \Theta, \Delta, \Sigma \vdash_{\mathcal{L}} \text{rec } t u v w : A} \text{ (Rec)}$$

Table 4: Type System for System  $\mathcal{L}_{\text{rec}}$

**Theorem 1 (Properties of reductions in System  $\mathcal{L}_{\text{rec}}$ )**

1. If  $\Gamma \vdash_{\mathcal{L}} t : T$  then  $\text{dom}(\Gamma) = \text{fv}(t)$ .
2. *Subject Reduction: Reductions preserve types.*
3. *Church-Rosser: System  $\mathcal{L}_{\text{rec}}$  is confluent.*
4. *Adequacy: If  $\vdash_{\mathcal{L}} t : T$  in System  $\mathcal{L}_{\text{rec}}$ , and  $t$  is a normal form, then:*

$$\begin{aligned} T = \mathbb{N} &\Rightarrow t = S(S \dots (S 0)) \\ T = A \otimes B &\Rightarrow t = \langle u, s \rangle \\ T = A \multimap B &\Rightarrow t = \lambda x. s \end{aligned}$$

5. *System  $\mathcal{L}_{\text{rec}}$  is not strongly normalising, even for typeable terms.*

**Proof:**

1. By induction on type derivations.
2. By induction on type derivations, using a substitution lemma as usual. We show the case where the term has the form  $\text{rec } \langle t, t' \rangle u v w$  (for the other cases, the proof is the same as for System  $\mathcal{L}$ ).

Assume  $\Gamma \vdash_{\mathcal{L}} \text{rec } \langle t, t' \rangle u v w : A$ . If the reduction takes place inside  $t, t', u, v$  or  $w$  the property follows directly by induction. If the reduction takes place at the root, there are two cases:

- (a)  $\text{rec } \langle 0, t' \rangle u v w \rightarrow u$  if  $\text{fv}(t'vw) = \emptyset$ . Then, by part 1,  $\text{dom}(\Gamma) = \text{fv}(\text{rec } \langle 0, t' \rangle u v w) = \text{fv}(u)$ . The type derivation may end with (Exchange), in which case the result is trivial, or with (Rec), in which case the derivation has conclusion  $\Gamma \vdash_{\mathcal{L}} \text{rec } \langle 0, t' \rangle u v w : A$  with premises:  $\vdash_{\mathcal{L}} \langle 0, t' \rangle : \mathbb{N} \otimes \mathbb{N}$ ,  $\Gamma \vdash_{\mathcal{L}} u : A$ ,  $\vdash_{\mathcal{L}} v : A \multimap A$ ,  $\vdash_{\mathcal{L}} w : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ . Therefore the property holds.

- (b)  $\text{rec } \langle S t, t' \rangle u v w \rightarrow v(\text{rec } (w\langle t, t' \rangle) u v w)$  if  $\text{fv}(vw) = \emptyset$ . Reasoning in a similar way, we note that when the type derivation ends with an application of the rule (Rec), it has conclusion  $\Gamma, \Delta \vdash_{\mathcal{L}} \text{rec } \langle St, t' \rangle u v w : A$  with premises  $\Gamma \vdash_{\mathcal{L}} \langle St, t' \rangle : \mathbb{N} \otimes \mathbb{N}$ ,  $\Delta \vdash_{\mathcal{L}} u : A$ ,  $\vdash_{\mathcal{L}} v : A \multimap A$ , and  $\vdash_{\mathcal{L}} w : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ . If  $\Gamma \vdash_{\mathcal{L}} \langle St, t' \rangle : \mathbb{N} \otimes \mathbb{N}$ , then we can deduce  $\Gamma \vdash_{\mathcal{L}} \langle t, t' \rangle : \mathbb{N} \otimes \mathbb{N}$ , therefore we have  $\Gamma \vdash_{\mathcal{L}} w\langle t, t' \rangle : \mathbb{N} \otimes \mathbb{N}$ . Thus we can obtain  $\Gamma, \Delta \vdash_{\mathcal{L}} \text{rec } (w\langle t, t' \rangle) u v w : A$ . From these we deduce  $\Gamma, \Delta \vdash_{\mathcal{L}} v(\text{rec } w\langle t, t' \rangle) u v w : A$  as required.
3. Confluence can be proved directly, using Martin-Löf's technique (as it was done for System  $\mathcal{L}$ , see [2]) or can be obtained as a consequence of Klop's theorem for orthogonal higher-order reductions [44].
4. By induction on  $t$ . If  $t = 0$ ,  $\lambda x.t'$  or  $\langle t_1, t_2 \rangle$ , then we are done. Otherwise:
- If  $t = S t$ , it follows by induction.
  - If  $t = \text{rec } t_0 t_1 t_2 t_3$ . Since  $t$  is in normal form, so are the terms  $t_i$ . Since  $t$  is typable,  $t_0$  must be a term of type  $\mathbb{N} \otimes \mathbb{N}$ , and by induction,  $t_0$  is a pair of numbers. But then one of the recursor rules applies (contradiction).

The cases of application and let are similar.

5. A non-terminating typable term (using the encoding of a fixpoint operator), is described in Section 3.3.

□

### 3.3 The computational power of System $\mathcal{L}_{\text{rec}}$

We now prove that System  $\mathcal{L}_{\text{rec}}$  is Turing complete. First note that although in the linear  $\lambda$ -calculus we are not able to discard arguments of functions, terms are consumed by reduction. The idea of erasing by consuming is related to the notion of Solvability (see [10], Chapter 8) as it relies on reduction to the identity. Using this technique, in [2, 4] it is shown that in System  $\mathcal{L}$  there is a general form of erasing. We next apply this technique to System  $\mathcal{L}_{\text{rec}}$ .

The term  $\mathcal{E}(t, A)$  defined below erases a term  $t$  of type  $A$ , under certain conditions. In the definition we use a function  $\mathcal{M}$  to build a term of a specific type ( $\mathcal{E}$  and  $\mathcal{M}$  are mutually recursive).

**Definition 1 (Erasing)** *If  $\Gamma \vdash_{\mathcal{L}} t : A$ , then  $\mathcal{E}(t, A)$  is defined as follows:*

$$\begin{aligned}
\mathcal{E}(t, \mathbb{N}) &= \text{rec } \langle t, 0 \rangle I I I \\
\mathcal{E}(t, A \otimes B) &= \text{let } \langle x, y \rangle = t \text{ in } \mathcal{E}(x, A)\mathcal{E}(y, B) \\
\mathcal{E}(t, A \multimap B) &= \mathcal{E}(t\mathcal{M}(A), B) \\
\text{and} \\
\mathcal{M}(\mathbb{N}) &= 0 \\
\mathcal{M}(A \otimes B) &= \langle \mathcal{M}(A), \mathcal{M}(B) \rangle \\
\mathcal{M}(A \multimap B) &= \lambda x. \mathcal{E}(x, A)\mathcal{M}(B)
\end{aligned}$$

**Theorem 2** 1. *If  $\Gamma \vdash_{\mathcal{L}} t : T$  then  $\Gamma \vdash_{\mathcal{L}} \mathcal{E}(t, T) : B \multimap B$ , for any type  $B$ .*

2.  *$\mathcal{M}(T)$  is closed and typeable:  $\vdash_{\mathcal{L}} \mathcal{M}(T) : T$ .*

3. *For any type  $T$ ,  $\mathcal{E}(\mathcal{M}(T), T) \rightarrow^* I$ .*

4.  *$\mathcal{M}(T)$  is normalisable.*

**Proof:** The first two parts are proved by simultaneous induction on  $T$ , as done for System  $\mathcal{L}$  [4]. The third part is proved by induction on  $T$ .

- If  $T = \mathbb{N}$ , then  $\mathcal{M}(T) = 0$ , and  $\mathcal{E}(0, \mathbb{N}) = \text{rec } \langle 0, 0 \rangle I I I \rightarrow I$ .

- If  $T = A \otimes B$ , then  $\mathcal{M}(A \otimes B) = \langle \mathcal{M}(A), \mathcal{M}(B) \rangle$ , then

$$\begin{aligned} \mathcal{E}(\langle \mathcal{M}(A), \mathcal{M}(B) \rangle, A \otimes B) &= \text{let } \langle x, y \rangle = \langle \mathcal{M}(A), \mathcal{M}(B) \rangle \text{ in } \mathcal{E}(x, A) \mathcal{E}(y, B) \\ &\rightarrow \mathcal{E}(\mathcal{M}(A), A) \mathcal{E}(\mathcal{M}(B), B) \xrightarrow{(\text{I.H.})} II \rightarrow I \end{aligned}$$

Note that, by induction,  $\mathcal{E}(\mathcal{M}(A), A) \rightarrow^* I$  and  $\mathcal{E}(\mathcal{M}(B), B) \rightarrow^* I$ .

- If  $T = A \multimap B$  then  $\mathcal{M}(T) = \lambda x. \mathcal{E}(x, A) \mathcal{M}(B)$ , therefore

$$\begin{aligned} &\mathcal{E}(\lambda x. \mathcal{E}(x, A) \mathcal{M}(B), A \multimap B) \\ &= \mathcal{E}((\lambda x. \mathcal{E}(x, A) \mathcal{M}(B)) \mathcal{M}(A), B) \\ &\rightarrow \mathcal{E}(\mathcal{E}(\mathcal{M}(A), A) \mathcal{M}(B), B) \\ &\xrightarrow{(\text{I.H.})} \mathcal{E}(I \mathcal{M}(B), B) \rightarrow \mathcal{E}(\mathcal{M}(B), B) \xrightarrow{(\text{I.H.})} I \end{aligned}$$

The last part is proved by induction on  $T$ . □

$\mathcal{L}_{\text{rec}}$ , unlike System  $\mathcal{L}$ , is not normalising, and there are terms that cannot be erased using this definition. There are even normalising terms that cannot be erased by reduction. For example, consider the following term  $Y_N$  which represents a fixpoint operator (more details are given in Section 5):

$$Y_N = \lambda f. \text{rec } \langle S(0), 0 \rangle 0 f (\lambda x. \text{let } \langle y, z \rangle = x \text{ in } \langle S(y), z \rangle)$$

This term is typable (it has type  $(N \multimap N) \multimap N$ ) and is a normal form (the recursor rules do not apply because  $f$  is a variable). However, the term

$$\mathcal{E}(Y_N, (N \multimap N) \multimap N) = \text{rec } \langle Y_N(\lambda x. \mathcal{E}(x, N) 0), 0 \rangle I I I$$

does not have a normal form. On the positive side, closed terms of type  $N$ , or tuples where the elements are terms of type  $N$ , can indeed be erased using this technique. Erasing “by consuming” reflects the work that needs to be done to effectively dispose of a data structure (where each component is garbage collected).

**Theorem 3** *Let  $T$  be a type generated by the grammar:  $A, B ::= N \mid A \otimes B$ . If  $\vdash_{\mathcal{L}} t : T$  and  $t$  has a normal form, then  $\mathcal{E}(t, T) \rightarrow^* I$ .*

**Proof:** By induction on  $T$ .

- If  $T = N$ , then  $\mathcal{E}(t, T) = \text{rec } \langle t, 0 \rangle I I I$ . Since  $t$  is normalising,  $t \rightarrow^* v$ , and by the Adequacy result (Theorem 1),  $v = S^n 0$ ,  $n \geq 0$ . Therefore  $\text{rec } \langle t, 0 \rangle I I I \rightarrow^* \text{rec } \langle S^n 0, 0 \rangle I I I \rightarrow^* I$ .
- If  $T = A \otimes B$ :  $\mathcal{E}(t, T) = \text{let } \langle x, y \rangle = t \text{ in } \mathcal{E}(x, A) \mathcal{E}(y, B)$ . Since  $t$  is normalisable then, by Adequacy (Theorem 1),  $t \rightarrow^* v = \langle u, s \rangle$ . Thus  $\text{let } \langle x, y \rangle = t \text{ in } \mathcal{E}(x, A) \mathcal{E}(y, B) \rightarrow^* \text{let } \langle x, y \rangle = \langle u, s \rangle \text{ in } \mathcal{E}(x, A) \mathcal{E}(y, B) \rightarrow \mathcal{E}(u, A) \mathcal{E}(s, B)$ . By induction hypothesis  $\mathcal{E}(u, A) \rightarrow^* I$  and  $\mathcal{E}(s, B) \rightarrow^* I$ , therefore  $\mathcal{E}(u, A) \mathcal{E}(s, B) \rightarrow^* II \rightarrow I$ .

□

There is also a mechanism to copy closed terms in  $\mathcal{L}_{\text{rec}}$ :

**Definition 2 (Duplication)** Define  $D^A : A \multimap A \otimes A$  as:

$$\lambda x. \text{rec } \langle S(S\ 0), 0 \rangle \langle \mathcal{M}(A), \mathcal{M}(A) \rangle F I$$

where  $F = (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, x \rangle)$ .

**Theorem 4** *If  $\vdash_{\mathcal{L}} t : A$  then  $D^A t \rightarrow^* \langle t, t \rangle$ .*



$$\frac{\frac{\Gamma \vdash_{\mathcal{L}} t : \mathbb{N}}{\Gamma \vdash_{\mathcal{L}} \langle t, 0 \rangle : \mathbb{N} \otimes \mathbb{N}} \quad \Theta \vdash_{\mathcal{L}} u : A \quad \Delta \vdash_{\mathcal{L}} v : A \multimap A \quad \vdash_{\mathcal{L}} I : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}}{\Gamma, \Theta, \Delta \vdash_{\mathcal{L}} \text{rec } \langle t, 0 \rangle u v I : A}$$

Figure 1: Type derivation for “iter”  $t u v$

**Proof:** By the definition of  $\rightarrow$ .

$$\begin{aligned}
D^A t &\rightarrow \text{rec } \langle S(S\ 0), 0 \rangle \langle \mathcal{M}(A), \mathcal{M}(A) \rangle (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle) I \\
&\rightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle)^2 \langle \mathcal{M}(A), \mathcal{M}(A) \rangle \\
&\rightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle)^2 \langle \mathcal{M}(A), \mathcal{M}(A) \rangle \\
&\rightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle) (\mathcal{E}(\mathcal{M}(A), A) \langle \mathcal{M}(A), t \rangle) \\
&\rightarrow^* (\lambda y. \text{let } \langle z, w \rangle = y \text{ in } \mathcal{E}(z, A) \langle w, t \rangle) \langle \mathcal{M}(A), t \rangle \\
&\rightarrow^* \mathcal{E}(\mathcal{M}(A), A) \langle t, t \rangle \rightarrow^* \langle t, t \rangle
\end{aligned}$$

□

The encoding of System  $\mathcal{L}$ ’s iterator, defined in Section 3.1, behaves as expected. System  $\mathcal{L}$  is a sub-system of  $\mathcal{L}_{\text{rec}}$ .

**Proposition 1**

$$\begin{aligned}
\text{“iter” } t u v &\rightarrow^* u && \text{if } t \rightarrow^* 0, \text{fv}(v) = \emptyset \\
\text{“iter” } t u v &\rightarrow^* v(\text{“iter” } t_1 u v) && \text{if } t \rightarrow^* S(t_1), \text{fv}(v) = \emptyset
\end{aligned}$$

**Proof:**

- If  $t \rightarrow^* 0$ :

$$\text{“iter” } t u v \stackrel{\text{def}}{=} \text{rec } \langle t, 0 \rangle u v I \rightarrow^* \text{rec } \langle 0, 0 \rangle u v I \rightarrow u, \text{ if } \text{fv}(v) = \emptyset$$

- If  $t \rightarrow^* S(t_1)$ :

$$\begin{aligned}
\text{“iter” } t u v &\stackrel{\text{def}}{=} \text{rec } \langle t, 0 \rangle u v I \rightarrow^* \text{rec } \langle S(t_1), 0 \rangle u v I \\
&\rightarrow v(\text{rec } I \langle t_1, 0 \rangle u v I), \text{ if } \text{fv}(t_1 v) = \emptyset \\
&\rightarrow v(\text{rec } \langle t_1, 0 \rangle u v I) \stackrel{\text{def}}{=} v(\text{“iter” } t_1 u v)
\end{aligned}$$

□

If  $\Gamma \vdash_{\mathcal{L}} t : \mathbb{N}$ ,  $\Theta \vdash_{\mathcal{L}} u : A$ , and  $\Delta \vdash_{\mathcal{L}} v : A \multimap A$ , then  $\Gamma, \Theta, \Delta \vdash_{\mathcal{L}} \text{rec } \langle t, 0 \rangle u v I : A$ , that is “iter”  $t u v$  is properly typed in System  $\mathcal{L}_{\text{rec}}$ , as shown in Figure 1.

**Corollary 1** System  $\mathcal{L}_{\text{rec}}$  has all the computation power of System  $\mathcal{L}$ , thus, any function definable in System  $\mathcal{T}$  can be defined in  $\mathcal{L}_{\text{rec}}$ .

We now show that the encoding of the minimiser given in Section 3.1 behaves as expected.

**Theorem 5 (Minimisation in System  $\mathcal{L}_{\text{rec}}$ )** Let  $\bar{f}$  be a closed  $\lambda$ -term in  $\mathcal{L}_{\text{rec}}$ , encoding the total function  $f$  on the natural numbers. The term  $M$  encodes  $\mu_f$ .

**Proof:** Consider the non-empty sequence  $S = f(i), f(i+1), \dots, f(i+n)$ , such that  $f(i+n)$  is the first element in the sequence that is equal to zero. Then

$$\text{rec } \langle \bar{f} S^i 0, S^i 0 \rangle 0 (\lambda x. S(x)) F \rightarrow^* S^n 0$$

We proceed by induction on the length of  $S$ .

- Basis:  $S = f(i)$ . Thus

$$\begin{array}{c} \text{rec } \langle \bar{f}(S^i 0), S^i 0 \rangle 0 (\lambda x. S(x)) F \\ \rightarrow^* \text{rec } \langle 0, S^i 0 \rangle 0 (\lambda x. S(x)) F \rightarrow 0 \end{array}$$

- Induction: If  $S = f(i), f(i+1), \dots, f(i+n)$ , then  $f(i) > 0$ , therefore  $\bar{f} \bar{i}$  reduces to a term of the form  $(S t)$ . Therefore

$$\begin{array}{c} \text{rec } \langle \bar{f}(S^i 0), S^i 0 \rangle 0 (\lambda x. S(x)) F \\ \rightarrow^* \text{rec } \langle S t, S^i 0 \rangle 0 (\lambda x. S(x)) F \\ \rightarrow^* S(\text{rec } \langle \bar{f}(S^{i+1} 0), S^{i+1} 0 \rangle 0 (\lambda x. S(x)) F) \\ \text{(I.H.)} \\ \rightarrow^* S(S^{n-1} 0) = S^n 0 \end{array}$$

Now, let  $j = \min\{x \in \mathbb{N} \mid f(x) = 0\}$ , and consider the sequence  $f(0), \dots, f(j)$ . Therefore  $\text{rec } \langle \bar{f} 0, 0 \rangle 0 (\lambda x. S(x)) F \rightarrow^* S^j 0$ . Note that, if there exists no  $x$  such that  $f(x) = 0$ , then  $\text{rec } \langle \bar{f} 0, 0 \rangle 0 (\lambda x. S(x)) F$  diverges, and so does the minimisation of  $f$ .  $\square$

**Corollary 2** *System  $\mathcal{L}_{\text{rec}}$  is Turing complete.*

### 3.4 Unbounded recursion vs. iteration and minimisation

There are two standard ways of extending the primitive recursive functions so that all partial recursive functions are obtained. One is unbounded minimisation, the other is unbounded recursion. For first-order functions (i.e., functions of type level 1), both methods are equivalent, see for instance [11]. Starting from System  $\mathcal{L}$  we could add a minimiser, with two reduction rules:

$$\begin{array}{ll} \mu 0 u f & \rightarrow u, & \text{fv}(f) = \emptyset \\ \mu (S t) u f & \rightarrow \mu (f (S u)) (S u) f & \text{fv}(f t u) = \emptyset \end{array}$$

and a typing rule

$$\frac{\Gamma \vdash_{\mathcal{L}} t : \mathbb{N} \quad \Theta \vdash_{\mathcal{L}} u : \mathbb{N} \quad \Delta \vdash_{\mathcal{L}} v : \mathbb{N} \multimap \mathbb{N}}{\Gamma, \Theta, \Delta \vdash_{\mathcal{L}} \mu t u v : \mathbb{N}} \text{ (Min)}$$

We will refer to this extension as System  $\mathcal{L}_{\mu}$ .

#### Theorem 6 (Properties of reductions in System $\mathcal{L}_{\mu}$ )

1. If  $\Gamma \vdash_{\mathcal{L}} t : T$  then  $\text{dom}(\Gamma) = \text{fv}(t)$ .
2. Subject Reduction: If  $\Gamma \vdash_{\mathcal{L}} t : T$  and  $t \rightarrow t'$  then  $\Gamma \vdash_{\mathcal{L}} t' : T$ .
3. System  $\mathcal{L}_{\mu}$  is confluent: If  $t \rightarrow^* u$  and  $t \rightarrow^* v$  then there is some term  $s$  such that  $u \rightarrow^* s$  and  $v \rightarrow^* s$ .

**Proof:**

1. By induction on the type derivation.
2. Straightforward extension of the proof given for System  $\mathcal{L}$  in [4], by induction on the type derivation  $\Gamma \vdash_{\mathcal{L}} t : T$ . We show the case where the term  $t$  is  $\mu s u f$  and there is a type derivation ending in:

$$\frac{\Gamma \vdash_{\mathcal{L}} s : \mathbb{N} \quad \Theta \vdash_{\mathcal{L}} u : \mathbb{N} \quad \Delta \vdash_{\mathcal{L}} f : \mathbb{N} \multimap \mathbb{N}}{\Gamma, \Theta, \Delta \vdash_{\mathcal{L}} \mu s u f : \mathbb{N}} \text{ (Min)}$$

If the reduction step takes place inside  $s$ ,  $u$  or  $f$ , the result follows directly by induction. If reduction takes place at the root, we have two cases:

- (a)  $\mu 0 u f \rightarrow u$ , with  $\text{fv}(f) = \emptyset$ . Note that  $\text{fv}(\mu 0 u f) = \text{fv}(u) = \text{dom}(\Theta)$  by part 1, and we have  $\Theta \vdash_{\mathcal{L}} u : \mathbb{N}$ .
- (b)  $\mu (S t) u f \rightarrow \mu (f (S u)) (S u) f$ , with  $\text{fv}(tuf) = \emptyset$ . Then  $\text{fv}(\mu (S t) u f) = \emptyset$ , and we have:

$$\frac{\vdash_{\mathcal{L}} S t : \mathbb{N} \quad \vdash_{\mathcal{L}} u : \mathbb{N} \quad \vdash_{\mathcal{L}} f : \mathbb{N} \multimap \mathbb{N}}{\vdash_{\mathcal{L}} \mu (S t) u f : \mathbb{N}} \text{ (Min)}$$

Therefore:

$$\frac{\frac{\vdash_{\mathcal{L}} f : \mathbb{N} \multimap \mathbb{N} \quad \vdash_{\mathcal{L}} S u : \mathbb{N}}{\vdash_{\mathcal{L}} f(S u) : \mathbb{N}} \quad \frac{\vdash_{\mathcal{L}} u : \mathbb{N}}{\vdash_{\mathcal{L}} S u : \mathbb{N}} \quad \vdash_{\mathcal{L}} f : \mathbb{N} \multimap \mathbb{N}}{\vdash_{\mathcal{L}} \mu (f(S u)) (S u) f : \mathbb{N}} \text{ (Min)}$$

3. Using Tait-Martin-Löf's method (see [10] for more details).

□

To show Turing completeness of System  $\mathcal{L}_{\mu}$  we need to show that we can encode unbounded minimisation.

First, we recall the following result from Kleene [41].

**Theorem 7 (The Kleene normal form)** *Let  $h$  be a partial recursive function on  $\mathbb{N}^k$ . Then, a number  $n$  and two primitive recursive functions  $f, g$  can be found such that  $h(x_1, \dots, x_k) = f(\mu_y(g(n, x_1, \dots, x_k, y)))h(x_1, \dots, x_k) = f(\mu_g(n, x_1, \dots, x_k))$ .*

Therefore we only have to prove that we can encode minimisation of primitive recursive functions, and we can rely on the fact that primitive recursive functions can be encoded in System  $\mathcal{L}$ . Below we give the encoding of minimisation for functions of arity 1 (the extension to functions of arity  $n > 1$  is straightforward).

**Theorem 8 (Unbounded minimisation in System  $\mathcal{L}_{\mu}$ )** *If  $f : \mathbb{N} \rightarrow \mathbb{N}$  is a primitive recursive function and  $\bar{f}$  is its encoding in System  $\mathcal{L}_{\mu}$ , then*

$$\mu_f = \mu (\bar{f} 0) 0 \bar{f}$$

**Proof:** Similar to the proof for System  $\mathcal{L}_{\text{rec}}$  (Theorem 5), considering the non-empty sequence  $S = f(i), f(i+1), \dots, f(j)$ , such that  $f(j)$  is the first element in the sequence that is equal to zero, and showing (by induction on the length of  $S$ ) that:

$$\mu (\bar{f} \bar{i}) \bar{i} \bar{f} \rightarrow^* \bar{j}.$$

□

**Corollary 3** *System  $\mathcal{L}_{\mu}$  is Turing complete.*

We can also encode System  $\mathcal{L}_{\text{rec}}$  into System  $\mathcal{L}_{\mu}$ , simulating the recursor with iter and  $\mu$ . Consider the following term:

$$f = \lambda n. pr_1(\text{iter } n \langle t, t' \rangle (w \circ \text{pred}_1))$$

where  $\text{pred}_1$  is such that  $\text{pred}_1 \langle S(t), t' \rangle = \langle t, t' \rangle$ . The function  $f$ , given  $n$ , will produce  $pr_1((w \circ \text{pred}_1)^n \langle t, t' \rangle)$ . Now consider  $(\mu t 0 f)$ , which will lead to the following sequence:

$$\mu t 0 f \rightarrow \mu f(1) 1 f \rightarrow \mu f(2) 2 f \rightarrow \mu f(3) 3 f \rightarrow \dots \rightarrow n$$

where  $n$  is the minimum number such that  $(w \circ \text{pred}_1)^n \langle t, t' \rangle$  produces  $\langle 0, t'' \rangle$ . Now, one can encode  $\text{rec } \langle t, t' \rangle u v w$  as:

$$\text{iter } (\mu t 0 f) u v$$

Notice that  $\text{rec } \langle St, t' \rangle u v w$  will iterate  $v$  until  $w(t, t')$  is equal to zero, and that  $\mu t 0 f$  will count the number of iterations that will actually be necessary, or will go on forever if that never happens.

$\mathcal{L}_{\text{rec}}$  can be seen as a more compact version of System  $\mathcal{L}_\mu$  where the recursor can perform both bounded iteration or minimisation. Note that, in recursion theory, the minimiser cannot replace bounded iteration as we now show.

**Lemma 1** *For any function  $f(x_1, \dots, x_n)$ ,  $n \geq 0$ , defined from the initial functions (0, S and projections) and composition, without using the primitive recursive scheme, there is a constant  $k$  such that  $f(x_1, x_2, \dots, x_n) = x_i + k$  or  $f(x_1, x_2, \dots, x_n) = k$ .*

**Proof:** By structural induction. The base cases are trivial. Let us consider the composition case:

1. The external function is 0. Then  $k = 0$ .
2. The external function is S. Then we have  $S(f(X))$  for some function  $f$  (recall that  $X$  is notation for  $x_1, \dots, x_n$ ). By the induction hypothesis  $f(X) = x_i + k$  or  $f(X) = k$ . In the first case we have  $S(f(X)) = x_i + (k + 1)$  and in second case we have  $S(f(X)) = k + 1$ .
3. The external function is  $pr_n$ . Then we have  $pr_n(f_1(X), \dots, f_n(X)) = f_i(X)$ . The result holds by the induction hypothesis for  $f_i(X)$ .
4. The external function  $g$  is itself defined by composition. Then, by induction hypothesis, either  $g(f_1(X), \dots, f_n(X)) = k$ , or  $g(f_1(X), \dots, f_n(X)) = f_i(X) + k$ , in which case the result follows by induction on  $f_i$ .

□

**Theorem 9** *Minimisation applied to functions in the previous class either returns 0 or is not defined.*

**Proof:** By the previous lemma, when  $f(x_1, \dots, x_n) = 0$ , then either it is the constant function returning 0, or it returns 0 when the argument  $x_i = 0$ . In the first case  $\mu_f$  returns 0, and in the second case either  $i = n$  and then  $\mu_f$  returns 0, or  $\mu_f$  diverges. □

Another way to obtain Turing completeness of typed  $\lambda$ -calculi is via fixpoint operators and conditionals, as done in PCF [53]. We study in Section 5 the relation between  $\mathcal{L}_{\text{rec}}$  and PCF.

## 4 Evaluation strategies for System $\mathcal{L}_{\text{rec}}$

In this section we define two evaluation strategies for System  $\mathcal{L}_{\text{rec}}$  and derive a stack-based abstract machine.

**Call-by-name** The CBN evaluation relation for closed terms in System  $\mathcal{L}_{\text{rec}}$  is defined in Table 5. The notation  $t \Downarrow V$  means that the closed term  $t$  evaluates in System  $\mathcal{L}_{\text{rec}}$  to the value  $V$ . Values are terms of the form 0,  $St$ ,  $\lambda x.t$  and  $\langle s, t \rangle$ , i.e., *weak head normal forms* (whnf). Note that System  $\mathcal{L}_{\text{rec}}$  does not evaluate under an S symbol, since S is used as a constructor for natural numbers.

The evaluation relation  $\cdot \Downarrow \cdot$  corresponds to *standard reduction* to weak head normal form. Recall that a reduction is called standard if the contraction of redexes is made from left-to-right (i.e., leftmost-outermost). It is well known that for the  $\lambda$ -calculus [10], the standard reduction is normalising, that is, if a term has a normal form, then it will be reached. A “standardisation” result holds for closed terms in  $\mathcal{L}_{\text{rec}}$ , as the following theorem shows.

**Theorem 10** *If  $\vdash_{\mathcal{L}} t : T$  (i.e.,  $t$  is a closed term in  $\mathcal{L}_{\text{rec}}$ ) and  $t$  has a whnf, then  $t \Downarrow V$ , for some  $V$ .*

$$\begin{array}{c}
\frac{V \text{ is a value}}{V \Downarrow V} \textit{Val} \quad \frac{s \Downarrow \lambda x.u \quad u[t/x] \Downarrow V}{s \ t \Downarrow V} \textit{App} \quad \frac{t \Downarrow \langle t_1, t_2 \rangle \quad (\lambda xy.u)t_1 t_2 \Downarrow V}{\textit{let } \langle x, y \rangle = t \textit{ in } u \Downarrow V} \textit{Let} \\
\frac{t \Downarrow \langle t_1, t_2 \rangle \quad t_1 \Downarrow 0 \quad u \Downarrow V}{\textit{rec } t \ u \ v \ w \Downarrow V} \textit{Rec1} \quad \frac{t \Downarrow \langle t_1, t_2 \rangle \quad t_1 \Downarrow S \ t' \quad v(\textit{rec } (w \langle t', t_2 \rangle)) \ u \ v \ w \Downarrow V}{\textit{rec } t \ u \ v \ w \Downarrow V} \textit{Rec2}
\end{array}$$

Table 5: CBN evaluation for System  $\mathcal{L}_{\text{rec}}$

**Proof:** We rely on Klop’s result [42, 20], which states that leftmost-outermost reduction is normalising for left-normal orthogonal Combinatory Reduction Systems (CRSs). A CRS is orthogonal if its rules are left-linear (i.e., the left hand-sides of the rewrite rules contain no duplicated variables) and non-overlapping (there are no critical pairs). A CRS is left-normal if on the left hand-sides of the rewrite rules, all the function symbols appear before the variables. The  $\lambda$ -calculus is an example of a left-normal orthogonal CRS, as is System  $\mathcal{L}_{\text{rec}}$ . Therefore, leftmost-outermost reduction is normalising for  $\mathcal{L}_{\text{rec}}$ . The result follows, since CBN performs leftmost-outermost reduction.  $\square$

For open terms, the set of weak head normal forms includes more kinds of terms, since, for instance, reduction of an application will be blocked if the argument is open. However, for a given open term one can consider all the free variables as constants and proceed with closed reduction as shown in [24] (see also [12]).

**Call-by-value** A call-by-value evaluation relation for System  $\mathcal{L}_{\text{rec}}$  can be obtained from the CBN relation by changing the rule for application, as usual.

$$\frac{s \Downarrow \lambda x.u \quad t \Downarrow V' \quad u[V'/x] \Downarrow V}{s \ t \Downarrow V}$$

There is no change in the *Rec* and *Let* rules, since they rely on the *App* rule. Unlike CBN, the CBV strategy does not always reach a value, even if a closed term has one (Theorem 10 does not hold for a CBV strategy). For example, recall the term  $Y_N$  in Section 3.3, and consider  $(\lambda xy.\textit{rec } \langle 0, 0 \rangle \ I \ \mathcal{E}(x, N) \ I)y)(Y_N I)$ . This term has a value under the CBN strategy, but not under CBV. In fact, innermost strategies are normalising in an orthogonal system if and only if the system is itself strongly normalising.

## 4.1 Stack machine for System $\mathcal{L}_{\text{rec}}$

Intermediate languages that incorporate linearity have well known implementation advantages whether in compilers, static analysis, or whenever resources are limited [47, 49, 13, 57]. Based on these previous works, we finish this section by illustrating how simply System  $\mathcal{L}_{\text{rec}}$  can be implemented as a stack machine. We show a call-by-name version, but it is straightforward to modify to other reduction strategies.

The basic principle of the machine is to find the next redex, using a stack  $\mathcal{S}$  to store future computations. The elements of the stack are terms in an extension of  $\mathcal{L}_{\text{rec}}$  which includes the following additional kinds of terms:  $LET(x, y, t)$ ,  $REC(u, v, w)$ ,  $REC(n, u, v, w)$ , where  $x, y$  are variables and  $n, t, u, v, w$  are  $\mathcal{L}_{\text{rec}}$  terms.

The configurations of the machine are pairs consisting of a term and a stack of extended terms. Unlike Krivine’s machine or its variants (see for instance [35, 17, 26]) we do not need to include an environment (sometimes called store, as in [57]) in the configurations. Indeed, the environment is used to store bindings for variables, but here as soon as a binding of a variable to a term is known we can replace the unique occurrence of that variable (the calculus is syntactically linear). In other words, instead of building an environment, we use “assignment” and replace the occurrence of the variable by the term.

The transitions of the machine are given in Table 6.

(app)	$(MN, \mathcal{S})$	$\Rightarrow$	$(M, N : \mathcal{S})$
(abs)	$(\lambda x.M, N : \mathcal{S})$	$\Rightarrow$	$(M[N/x], \mathcal{S})$
(let)	$(\text{let } \langle x, y \rangle = N \text{ in } M, \mathcal{S})$	$\Rightarrow$	$(N, LET(x, y, M) : \mathcal{S})$
(pair1)	$(\langle N_1, N_2 \rangle, LET(x, y, M) : \mathcal{S})$	$\Rightarrow$	$(M[N_1/x][N_2/y], \mathcal{S})$
(rec)	$(\text{rec } N \ U \ V \ W, \mathcal{S})$	$\Rightarrow$	$(N, REC(U, V, W) : \mathcal{S})$
(pair2)	$(\langle N_1, N_2 \rangle, REC(U, V, W) : \mathcal{S})$	$\Rightarrow$	$(N_1, REC'(N_2, U, V, W) : \mathcal{S})$
(zero)	$(0, REC'(T, U, V, W) : \mathcal{S})$	$\Rightarrow$	$(U, \mathcal{S})$
(succ)	$(S(N), REC'(T, U, V, W) : \mathcal{S})$	$\Rightarrow$	$(V, (\text{rec } (W \langle N, T \rangle)) \ U \ V \ W) : \mathcal{S})$

Table 6: Stack machine for System  $\mathcal{L}_{\text{rec}}$

For a program (closed term  $M$ ), the machine is started with an empty stack:  $(M, [])$ . The machine stops when no rule can apply.

The use of “assignment” means that there is no manipulation (no copying, erasing, or even searching for bindings) in environments usually associated to these kinds of implementations.

The correctness of the machine with respect to the CBN evaluation relation is proved by induction in the usual way.

**Theorem 11** *If  $\vdash_{\mathcal{L}} t : T$  and there is a value  $V$  such that  $t \Downarrow V$ , then  $(t, []) \Rightarrow^* (V, [])$ .*

**Proof:** By induction on the evaluation relation, using Subject Reduction (Theorem 1) and the following property:

If  $(t, \mathcal{S}) \Rightarrow (t', \mathcal{S}')$  then  $(t, \mathcal{S} \circ \mathcal{S}'') \Rightarrow (t', \mathcal{S}' \circ \mathcal{S}'')$ .

This property is proved by induction on  $(t, \mathcal{S})$ . Intuitively, since only the top of the stack is used to select a transition, it is clear that appending elements at the bottom of the stack does not affect the computation.  $\square$

## 5 Applications: Fixpoint operators and PCF

### 5.1 The role of conditionals

Recursive function definitions based on fixpoint operators rely on the use of a non-linear conditional that should discard the branch corresponding to an infinite computation. For instance, the definition of factorial:

$$\text{fact} = Y(\lambda f n. \text{cond } n \ 1 \ (n * f(n-1)))$$

relies on the fact that `cond` will return 1 when the input number is 0, and discard the non-terminating “else” branch. Enabling the occurrence of the (bound) variable, used to iterate the function ( $f$  in the above definition), in only one branch of the conditional is crucial for the definition of interesting recursive programs. This is why denotational linear versions of PCF [52] allow stable variables to be used non-linearly but not to be abstracted, since their only purpose is to obtain fixpoints.

Fixpoint operators can be encoded in System  $\mathcal{L}_{\text{rec}}$ : recall the term  $Y_{\mathbb{N}}$  in Section 3.3. More generally, for any type  $A$  we define the term

$$Y_A = \lambda f. \text{rec } \langle S(0), 0 \rangle \ \mathcal{M}(A) \ f \ W$$

where  $W$  represents the term  $(\lambda x. \text{let } \langle y, z \rangle = x \text{ in } \langle S(y), z \rangle)$ . For every type  $A$ ,  $Y_A : (A \multimap A) \multimap A$  is well-typed in System  $\mathcal{L}_{\text{rec}}$  (see Figure 2). Note that, for any closed term  $f$  of type  $A \multimap A$ , we have:

$$\begin{aligned} Y_A f &= \text{rec } \langle S(0), 0 \rangle \ \mathcal{M}(A) \ f \ W \\ &\rightarrow^* f(\text{rec } (\text{let } \langle y, z \rangle = \langle 0, 0 \rangle \text{ in } \langle S(y), z \rangle) \ \mathcal{M}(A) \ f \ W) \\ &\rightarrow f(\text{rec } \langle S(0), 0 \rangle \ \mathcal{M}(A) \ f \ W) = f(Y_A f) \end{aligned}$$

$$\frac{\frac{\vdots}{\vdash_{\mathcal{L}} \langle S(0), 0 \rangle : \mathbb{N} \otimes \mathbb{N}} \quad \frac{\vdots}{\vdash_{\mathcal{L}} \mathcal{M}(A) : A} \quad f : A \multimap A \vdash_{\mathcal{L}} f : A \multimap A \quad \frac{\vdots}{\vdash_{\mathcal{L}} W : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}}}{\frac{f : A \multimap A \vdash_{\mathcal{L}} \text{rec } \langle S(0), 0 \rangle \mathcal{M}(A) f W : A}{\vdash_{\mathcal{L}} \lambda f. \text{rec } \langle S(0), 0 \rangle \mathcal{M}(A) f W : (A \multimap A) \multimap A}}$$

Figure 2: Type derivation for  $Y_A$

Although  $Y_A$  behaves like a fixpoint operator, one cannot write useful recursive programs using fixpoint operators alone (i.e. without a conditional): if we apply  $Y_A$  to a linear function  $f$ , we obtain a non-normalisable term (recall the example in Section 3.3). Instead, in System  $\mathcal{L}_{\text{rec}}$ , recursive functions, such as factorial, can be easily encoded using **rec**:

$$\lambda n. \text{pr}_2(\text{rec } \langle n, 0 \rangle \langle S(0), S(0) \rangle (\lambda x. \text{let } \langle t, u \rangle = x \text{ in } F) I)$$

where  $F = \text{let } \langle t_1, t_2 \rangle = D^N t \text{ in } \langle S t_1, \text{mult } u t_2 \rangle$  and  $D^N$  is the duplicator term defined previously (see Definition 2). Note that, although conditionals are not part of System  $\mathcal{L}_{\text{rec}}$  syntax, reduction rules for **rec** use pattern-matching. In the remainder of this section we show how we can encode in System  $\mathcal{L}_{\text{rec}}$  recursive functions defined using fixpoints.

## 5.2 Encoding PCF in System $\mathcal{L}_{\text{rec}}$

PCF (Programming Language for Computable Functions) [53] can be seen as a minimalistic typed functional programming language. It is an extension of the simply typed  $\lambda$ -calculus with numbers, a fixpoint operator, and a conditional. Let us first recall its syntax. PCF is a variant of the typed  $\lambda$ -calculus, with a basic type  $\mathbb{N}$  for numbers and the following constants:

- $n : \mathbb{N}$ , for  $n = 0, 1, 2, \dots$
- $\text{succ}, \text{pred} : \mathbb{N} \rightarrow \mathbb{N}$
- $\text{iszero} : \mathbb{N} \rightarrow \mathbb{N}$ , such that

$$\begin{aligned}
\text{iszero } 0 &\rightarrow 0 \\
\text{iszero } (n + 1) &\rightarrow 1
\end{aligned}$$

- for each type  $A$ ,  $\text{cond}_A : \mathbb{N} \rightarrow A \rightarrow A \rightarrow A$ , such that

$$\begin{aligned}
\text{cond}_A 0 u v &\rightarrow u \\
\text{cond}_A (n + 1) u v &\rightarrow v
\end{aligned}$$

- for each type  $A$ ,  $Y_A : (A \rightarrow A) \rightarrow A$ , such that  $Y_A f \rightarrow f(Y_A f)$ .

**Definition 3** *PCF types and environments are translated into System  $\mathcal{L}_{\text{rec}}$  types using  $\langle \cdot \rangle$ :*

$$\begin{aligned}
\langle \mathbb{N} \rangle &= \mathbb{N} \\
\langle A \rightarrow B \rangle &= \langle A \rangle \multimap \langle B \rangle \\
\langle x_1 : T_1, \dots, x_n : T_n \rangle &= x_1 : \langle T_1 \rangle, \dots, x_n : \langle T_n \rangle
\end{aligned}$$

Since System  $\mathcal{L}_{\text{rec}}$  is Turing complete, it is clear that any PCF program can be encoded in System  $\mathcal{L}_{\text{rec}}$ . We define below an encoding in System  $\mathcal{L}_{\text{rec}}$  for terms in PCF, inspired by the encoding of System  $\mathcal{T}$  [4]. For convenience, we make the following abbreviations, where the variables  $x_1$  and  $x_2$  are assumed fresh, and  $[x]t$  is defined below:

$$\begin{aligned}
C_{x:A}^{x_1, x_2} t &= \text{let } \langle x_1, x_2 \rangle = D^A x \text{ in } t \\
A_y^x t &= ([x]t)[y/x]
\end{aligned}$$

$\langle n \rangle$	$=$	$S^n 0$	
$\langle \text{succ} \rangle$	$=$	$\lambda n. \text{rec } \langle n, 0 \rangle (S \ 0) (\lambda x. Sx) \ I$	
$\langle \text{pred} \rangle$	$=$	$\lambda n. pr_1(\text{rec } \langle n, 0 \rangle \langle 0, 0 \rangle (\lambda x. \text{let } \langle t, u \rangle = D^N(pr_2 \ x) \text{ in } \langle t, S \ u \rangle) \ I)$	
$\langle \text{iszero} \rangle$	$=$	$\lambda n. pr_1(\text{rec } \langle n, 0 \rangle \langle 0, S \ 0 \rangle (\lambda x. D^N(pr_2 \ x)) \ I)$	
$\langle Y_A \rangle$	$=$	$\lambda f. \text{rec } \langle S(0), 0 \rangle \mathcal{M}(\langle A \rangle) \ f \ (\lambda x. \text{let } \langle y, z \rangle = x \text{ in } \langle S(y), z \rangle)$	
$\langle \text{cond}_A \rangle$	$=$	$\lambda tuv. \text{rec } \langle t, 0 \rangle \ u \ (\lambda x. (\text{rec } \langle 0, 0 \rangle \ I \ \mathcal{E}(x, \langle A \rangle) \ I) v) \ I$	
$\langle x \rangle$	$=$	$x$	
$\langle uv \rangle$	$=$	$\langle u \rangle \langle v \rangle$	
$\langle \lambda x^A. t \rangle$	$=$	$\begin{cases} \lambda x. [x^A] \langle t \rangle & \text{if } x \in \text{fv}(t) \\ \lambda x. (\text{rec } \langle 0, 0 \rangle \ I \ \lambda y. \mathcal{E}(\mathcal{E}(y, \langle B \rangle \multimap \langle B \rangle) x, \langle A \rangle) \ I) \langle t \rangle & \text{otherwise} \end{cases}$	

Table 7: PCF compilation into  $\mathcal{L}_{\text{rec}}$

**Definition 4** Let  $t$  be a PCF term such that  $\text{fv}(t) = \{x_1, \dots, x_n\}$  and  $x_1 : A_1, \dots, x_n : A_n \vdash t : A$ . The compilation into System  $\mathcal{L}_{\text{rec}}$ , is defined as:  $[x_1^{A_1}] \dots [x_n^{A_n}] \langle t \rangle^1$ , where  $\langle \cdot \rangle$  is defined in Table 7, and for a term  $t$  and a variable  $x$ , such that  $x \in \text{fv}(t)$ ,  $[x]t$  is inductively defined in the following way:

$$\begin{aligned}
[x](S \ u) &= S([x]u) \\
[x]x &= x \\
[x](\lambda y. u) &= \lambda y. [x]u \\
[x^A](su) &= \begin{cases} C_{x:A}^{x_1, x_2} (A_y^{x_1} s) (A_y^{x_2} u) & x \in \text{fv}(s) \cap \text{fv}(u) \\ ([x]s)u & x \notin \text{fv}(u) \\ s([x]u) & x \notin \text{fv}(s) \end{cases}
\end{aligned}$$

Notice that  $[x]t$  is not defined for the entire syntax of System  $\mathcal{L}_{\text{rec}}$ . The reason for this is that, although other syntactic constructors (like recursors or pairs) may appear in  $t$ , they are the outcome of  $\langle \cdot \rangle$  and therefore are closed terms, where  $x$  does not occur free.

Note that  $\text{succ}$  is not encoded as  $\lambda x. Sx$ , since  $\mathcal{L}_{\text{rec}}$  does not evaluate under  $\lambda$  or  $S$ . We should not encode a divergent PCF program into a terminating term in  $\mathcal{L}_{\text{rec}}$ . In particular, the translation of  $\text{cond}_A (\text{succ}(Y_N I)) \ P \ Q$  is  $\langle \text{cond}_A \rangle (\langle \text{succ} \rangle (\langle Y_N \rangle I)) \langle P \rangle \langle Q \rangle$ , which diverges (if we encode  $\text{succ}$  as  $\lambda x. Sx$ , then we obtain  $\langle Q \rangle$ , which is not right).

Regarding abstractions or conditionals, the encoding is different from the one used in for System  $\mathcal{T}$  in [4]. We cannot use the same encoding as in System  $\mathcal{L}$ , where terms are erased by “consuming them”, because PCF, unlike System  $\mathcal{T}$ , is not strongly normalising. The technique used here for erasing could have been used for System  $\mathcal{L}$ , but erasing “by consuming” is preferred when possible (it reflects the work needed to erase a data structure).

Also note that the second case in the encoding for abstractions (see Table 7) uses a recursor on zero to discard the argument, where the function parameter is  $\lambda y. \mathcal{E}(\mathcal{E}(y, \langle B \rangle \multimap \langle B \rangle) x, \langle A \rangle)$ . The reason for this is that one cannot use  $x$  directly as the function parameter because that might make the term untypable, and just using  $\mathcal{E}(x, \langle A \rangle)$  would make the types work, but could encode strongly normalisable terms into terms with infinite reduction sequences (because  $\mathcal{E}(x, \langle A \rangle)$  might not terminate). For example, consider the encoding of  $(\lambda xy. y) Y_N$ .

The translation of a typable PCF term is also typable in System  $\mathcal{L}_{\text{rec}}$  (this is proved below). In particular, for any type  $A$ , the term  $\langle \text{cond}_A \rangle$  is well-typed. In Figure 3, we show the type derivation for the encoding of the conditional (we use  $V$  to represent the term  $\lambda x. (\text{rec } \langle 0, 0 \rangle \ I \ \mathcal{E}(x, \langle A \rangle) \ I) v$ ).

The type derivation for  $V$  depends on the fact that, if  $\Gamma \vdash_{\mathcal{L}} t : A$ , then for any type  $B$ , we have  $\Gamma \vdash_{\mathcal{L}} \mathcal{E}(t, A) : B \multimap B$  by Theorem 2. Note that the recursor on  $\langle 0, 0 \rangle$  in  $V$  discards the remaining recursion (corresponding to the branch of the conditional that is not needed), returning  $Iv$ .

We prove that the encoding respects types by induction. To make the induction work, we need to define an intermediate system where certain variables (not yet affected by the encoding) may

<sup>1</sup>We omit the types of variables when they do not play a role in the compilation.



$$\begin{array}{c}
t : \mathbf{N} \vdash_{\mathcal{L}} \langle t, 0 \rangle : \mathbf{N} \otimes \mathbf{N} \quad u : \langle A \rangle \vdash_{\mathcal{L}} u : \langle A \rangle \quad v : \langle A \rangle \vdash_{\mathcal{L}} V : \langle A \rangle \multimap \langle A \rangle \quad \vdash_{\mathcal{L}} I : \mathbf{N} \otimes \mathbf{N} \multimap \mathbf{N} \otimes \mathbf{N} \\
\hline
t : \mathbf{N}, u : \langle A \rangle, v : \langle A \rangle \vdash_{\mathcal{L}} \text{rec } \langle t, 0 \rangle u V I : \langle A \rangle \\
\vdots \\
\vdash_{\mathcal{L}} \text{cond}_A : \mathbf{N} \multimap \langle A \rangle \multimap \langle A \rangle \multimap \langle A \rangle
\end{array}$$

Figure 3: Type derivation for  $\text{cond}_A$

**Axiom and Structural Rule:**

$$\begin{array}{c}
\frac{}{x : A \vdash_{\mathcal{L}_{\text{rec}}}^+ x : A} \text{(Axiom)} \quad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ t : C} \text{(Exchange)} \\
\\
\frac{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : B \quad \text{and } x \in X}{\Gamma, x : A \vdash_{\mathcal{L}_{\text{rec}}}^+ t : B} \text{(Weakening)} \quad \frac{\Gamma, x : A, x : A \vdash_{\mathcal{L}_{\text{rec}}}^+ t : B \quad \text{and } x \in X}{\Gamma, x : A \vdash_{\mathcal{L}_{\text{rec}}}^+ t : B} \text{(Contraction)}
\end{array}$$

**Logical Rules:**

$$\begin{array}{c}
\frac{\Gamma, x : A \vdash_{\mathcal{L}_{\text{rec}}}^+ t : B}{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ \lambda x. t : A \multimap B} (\multimap \text{Intro}) \quad \frac{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : A \multimap B \quad \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ u : A}{\Gamma, \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ (X_1 \cup X_2) tu : B} (\multimap \text{Elim}) \\
\\
\frac{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : A \quad \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ u : B}{\Gamma, \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ (X_1 \cup X_2) \langle t, u \rangle : A \otimes B} (\otimes \text{Intro}) \quad \frac{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : A \otimes B \quad x : A, y : B, \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ u : C}{\Gamma, \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ (X_1 \cup X_2) \text{let } \langle x, y \rangle = t \text{ in } u : C} (\otimes \text{Elim})
\end{array}$$

**Numbers:**

$$\begin{array}{c}
\frac{}{\vdash_{\mathcal{L}_{\text{rec}}}^+ 0 : \mathbf{N}} \text{(Zero)} \quad \frac{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : \mathbf{N}}{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ S(t) : \mathbf{N}} \text{(Succ)} \\
\\
\frac{\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : \mathbf{N} \otimes \mathbf{N} \quad \Theta \vdash_{\mathcal{L}_{\text{rec}}}^+ u : A \quad \Delta \vdash_{\mathcal{L}_{\text{rec}}}^+ v : A \multimap A \quad \Sigma \vdash_{\mathcal{L}_{\text{rec}}}^+ w : \mathbf{N} \otimes \mathbf{N} \multimap \mathbf{N} \otimes \mathbf{N}}{\Gamma, \Theta, \Delta, \Sigma \vdash_{\mathcal{L}_{\text{rec}}}^+ (X_1 \cup X_2 \cup X_3 \cup X_4) \text{rec } t u v w : A} \text{(Rec)}
\end{array}$$

Table 8: Typing rules for System  $\mathcal{L}_{\text{rec}}^{+X}$

occur non-linearly. More precisely, we consider an extension to System  $\mathcal{L}_{\text{rec}}$ , which allows variables on a certain set  $X$  to appear non-linearly in a term. We call the extended system System  $\mathcal{L}_{\text{rec}}^{+X}$ ; it is defined by the rules in Table 8. Intuitively, if  $X$  is the set of free-variables of  $t$ , then  $\langle t \rangle$  will be a System  $\mathcal{L}_{\text{rec}}$  term, except for the variables  $X = \text{fv}(t)$ , which may occur non-linearly, and  $[x_1] \dots [x_n] \langle t \rangle$ , will be a typed System  $\mathcal{L}_{\text{rec}}$  term. We can prove the following results regarding System  $\mathcal{L}_{\text{rec}}^{+X}$ .

**Lemma 2** *If  $\Gamma \vdash_{\mathcal{L}_{\text{rec}}}^+ t : A$ , where  $\text{dom}(\Gamma) = \text{fv}(t)$  and  $x \in X \subseteq \text{fv}(t)$ , then  $\Gamma \vdash_{\mathcal{L}_{\text{rec}}^{+X'}} [x]t : A$ , where  $X' = X \setminus \{x\}$ .*

**Proof:** By induction on  $t$ , using the fact that  $x : A \vdash_{\mathcal{L}_{\text{rec}}}^+ D^A x : A \otimes A$ . We show the cases for variable and application.

- $t \equiv x$ . Then  $[x]x = x$ , and using the axiom we obtain both  $x : A \vdash_{\mathcal{L}_{\text{rec}}^{+\{x\}}} x : A$  and  $x : A \vdash_{\mathcal{L}_{\text{rec}}^{+\emptyset}} x : A$ .
- $t \equiv uv$ , and  $x \in \text{fv}(u)$ ,  $x \notin \text{fv}(v)$  (the case where  $x \notin \text{fv}(u)$ ,  $x \in \text{fv}(v)$  is similar). Then  $[x]uv = ([x]u)v$  and  $\Gamma \vdash_{\mathcal{L}_{\text{rec}}^{+X}} uv : A$ . Let  $\Gamma_1 = \Gamma|_{\text{fv}(u)}$  and  $\Gamma_2 = \Gamma|_{\text{fv}(v)}$ . Then  $\Gamma_1 \vdash_{\mathcal{L}_{\text{rec}}^{+X}} u :$

$\frac{V \text{ is a value}}{V \Downarrow_{\text{PCF}} V}$	$\frac{s \Downarrow_{\text{PCF}} V' \quad V' t \Downarrow_{\text{PCF}} V \quad s \text{ is not a value}}{s t \Downarrow_{\text{PCF}} V}$	$\frac{u[t/x] \Downarrow_{\text{PCF}} V}{(\lambda x. u) t \Downarrow_{\text{PCF}} V}$
$\frac{t \Downarrow_{\text{PCF}} 0}{\text{pred } t \Downarrow_{\text{PCF}} 0}$	$\frac{t \Downarrow_{\text{PCF}} n+1}{\text{pred } t \Downarrow_{\text{PCF}} n}$	$\frac{t \Downarrow_{\text{PCF}} n}{\text{succ } t \Downarrow_{\text{PCF}} n+1}$
$\frac{t \Downarrow_{\text{PCF}} 0 \quad u \Downarrow_{\text{PCF}} V}{\text{cond}_A t u v \Downarrow_{\text{PCF}} V}$	$\frac{t \Downarrow_{\text{PCF}} n+1 \quad v \Downarrow_{\text{PCF}} V}{\text{cond}_A t u v \Downarrow_{\text{PCF}} V}$	$\frac{f(Y_A f) \Downarrow_{\text{PCF}} V}{Y_A f \Downarrow_{\text{PCF}} V}$
$\frac{t \Downarrow_{\text{PCF}} 0}{\text{iszero } t \Downarrow_{\text{PCF}} 0}$	$\frac{t \Downarrow_{\text{PCF}} n+1}{\text{iszero } t \Downarrow_{\text{PCF}} 1}$	

Table 9: CBN evaluation for PCF

$B \multimap A$  and  $\Gamma_2 \vdash_{\mathcal{L}_{\text{rec}}^{+x}} v : B$ , where  $\Gamma_1$  and  $\Gamma_2$  can only share variables in  $X$ . By induction hypothesis  $\Gamma_1 \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} [x]u : B \multimap A$ . Also, since  $x \notin \text{fv}(v)$  and  $\text{dom}(\Gamma_2) = \text{fv}(v)$ , we have  $\Gamma_2 \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} v : B$ . Therefore  $\Gamma \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} (x[u])v : A$ .

- $t \equiv uv$ ,  $x \in \text{fv}(u)$ , and  $x \in \text{fv}(v)$ . Let  $\Gamma_1 = \Gamma_{|\text{fv}(u) \setminus \{x\}}$  and  $\Gamma_2 = \Gamma_{|\text{fv}(v) \setminus \{x\}}$  and assume  $C$  is the type associated to  $x$  in  $\Gamma$ . Then  $\Gamma_1, x : C \vdash_{\mathcal{L}_{\text{rec}}^{+x}} u : B \multimap A$  and  $\Gamma_2, x : C \vdash_{\mathcal{L}_{\text{rec}}^{+x}} v : B$ . By induction hypothesis  $\Gamma_1, x : C \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} [x]u : B \multimap A$ , and  $\Gamma_2, x : C \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} [x]v : B$ . Thus  $\Gamma_1, x_1 : C \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} ([x]u)[x_1/x] : B \multimap A$ , and  $\Gamma_2, x_2 : C \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} ([x]v)[x_2/x] : B$ . Therefore  $\Gamma_1, x_1 : C, \Gamma_2, x_2 : C \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} (A_{x_1}^x u)(A_{x_2}^x v) : A$ . Also  $x : C \vdash_{\mathcal{L}_{\text{rec}}^{+\emptyset}} Dx : C \otimes C$ , therefore  $\Gamma_1, \Gamma_2, x : C \vdash_{\mathcal{L}_{\text{rec}}^{+x'}} \text{let } \langle x_1, x_2 \rangle = Dx \text{ in } (A_{x_1}^x u)(A_{x_2}^x v) : A$ .

□

**Lemma 3** *If  $\Gamma \vdash_{\text{PCF}} t : A$ , then  $\langle \Gamma_{|\text{fv}(t)} \rangle \vdash_{\mathcal{L}_{\text{rec}}^{+\text{fv}(t)}} \langle t \rangle : \langle A \rangle$ .*

**Proof:** By induction on the type derivation  $\Gamma \vdash_{\text{PCF}} t : A$ , as done for System  $\mathcal{T}$  in [4].

□

**Theorem 12** *If  $t$  is a PCF term of type  $A$  under a set of assumptions  $\Gamma$  for its free variables  $\{x_1, \dots, x_n\}$ , then  $\langle \Gamma_{|\text{fv}(t)} \rangle \vdash_{\mathcal{L}} [x_1] \dots [x_n] \langle t \rangle : \langle A \rangle$*

**Proof:** By induction on the number of free variables of  $t$ , using Lemmas 2 and 3.

□

Using the encodings given above, it is possible to simulate the evaluation of a PCF program in System  $\mathcal{L}_{\text{rec}}$ . More precisely, if  $t$  is a closed PCF term of type  $\mathbb{N}$ , which evaluates to  $v$  under a CBN semantics for PCF [53], then the encoding of  $t$  reduces in System  $\mathcal{L}_{\text{rec}}$  to the encoding of  $v$ , and evaluates under a CBN semantics to a value which is equal to the encoding of  $v$ . In Table 9 we recall the CBN rules for PCF:  $t \Downarrow_{\text{PCF}} V$  means that the closed term  $t$  evaluates to the value  $V$  (a value is either a number, a  $\lambda$ -abstraction, a constant, or a partially applied conditional).

**Lemma 4 (Substitution)** *Let  $t$  be a term in System  $\mathcal{L}_{\text{rec}}$ .*

1. *If  $x \in \text{fv}(t)$ , and  $\text{fv}(u) = \emptyset$ , then  $\langle t \rangle[\langle u \rangle/x] = \langle t[u/x] \rangle$*
2. *If  $x \in \text{fv}(t)$ , then  $([x]t)[u/x] \rightarrow^* t[u/x]$ .*

**Proof:** By induction on  $t$ .

□

**Lemma 5** *Let  $t$  be a closed PCF term. If  $t \Downarrow_{\text{PCF}} V$ , then  $\langle t \rangle \rightarrow^* \langle V \rangle$ .*

**Proof:** By induction on the evaluation relation, using a technique similar to the one used for System  $\mathcal{T}$  in [4]. Here we show the main steps of reduction for  $\text{cond}_A t u v$  where  $u, v$  are closed terms by assumption.

- If  $t \Downarrow_{\text{PCF}} 0$ :

$$\begin{aligned}
\langle \text{cond}_A t u v \rangle &= \langle \text{cond}_A \rangle \langle t \rangle \langle u \rangle \langle v \rangle \\
&\xrightarrow{*} \text{cond}_A 0 \langle u \rangle \langle v \rangle \\
&\xrightarrow{*} \langle u \rangle \xrightarrow{*} \langle V \rangle
\end{aligned}$$

- If  $t \Downarrow_{\text{PCF}} n+1$ , let  $v'$  be the term  $(\lambda x. (\text{rec } \langle 0, 0 \rangle I \mathcal{E}(x, A) I) \langle v \rangle)$ :

$$\begin{aligned}
\langle \text{cond}_A t u v \rangle &= \langle \text{cond}_A \rangle \langle S^{n+1} 0 \rangle \langle u \rangle \langle v \rangle \\
&\xrightarrow{*} \text{rec } \langle S^{n+1} 0, 0 \rangle \langle u \rangle v' I \\
&\xrightarrow{*} I \langle v \rangle \rightarrow \langle v \rangle \xrightarrow{*} \langle V \rangle.
\end{aligned}$$

For application, we rely on the substitution lemmas above. Note that for an application  $uv$ , where  $u$  is a constant, we rely on the correctness of the encodings for constants, which can be easily prove by induction. For example, in the case of **succ** it is trivial to prove that, if  $t$  is a number  $S^n 0$  in  $\mathcal{L}_{\text{rec}}$  ( $n \geq 0$ ), then  $\text{rec } \langle t, 0 \rangle (S 0) (\lambda x. Sx) I \rightarrow^* S^{n+1} 0$ .  $\square$

**Theorem 13** *Let  $t$  be a closed PCF term. If  $t \Downarrow_{\text{PCF}} V$ , then  $\exists V'$  such that  $\langle t \rangle \Downarrow V'$ , and  $V' =_{\mathcal{L}_{\text{rec}}} \langle V \rangle$ .*

**Proof:** By Lemma 5,  $t \Downarrow_{\text{PCF}} V$  implies  $\langle t \rangle \rightarrow^* \langle V \rangle$ . By Theorem 10,  $\langle t \rangle \Downarrow V'$ . Therefore, since  $\Downarrow \subset \rightarrow^*$  and the system is confluent (Theorem 1),  $V' =_{\mathcal{L}_{\text{rec}}} \langle V \rangle$ .  $\square$

**Lemma 6** *If  $t \Downarrow V$  and  $t =_{\mathcal{L}_{\text{rec}}} u$ , then  $u \Downarrow V'$  and  $V =_{\mathcal{L}_{\text{rec}}} V'$ .*

**Proof:** By transitivity of the equality relation.  $\square$

**Theorem 14** *Let  $t$  be a closed PCF term. If  $\langle t \rangle \Downarrow V$ , then  $\exists V'$ , such that,  $t \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$ .*

**Proof:** By induction on the evaluation relation, using Lemma 6. Note that, if  $t$  is a value different from a partially applied conditional, the result follows because  $t = V'$  and  $\langle t \rangle$  is also a value, i.e.  $\langle t \rangle = V$ , therefore  $\langle t \rangle = \langle V' \rangle = V$ . If  $t$  is an application  $uv$  then  $\langle t \rangle = \langle u \rangle \langle v \rangle$ , therefore  $\langle u \rangle \langle v \rangle \Downarrow V$  if  $\langle u \rangle \Downarrow \lambda x. s$  and  $s[\langle v \rangle/x] \Downarrow V$ . If  $\langle u \rangle \Downarrow \lambda x. s$ , then by I.H.  $u \Downarrow_{\text{PCF}} W$ , and  $\langle W \rangle =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ . Note that  $W$  is a value of arrow type, which compilation equals an abstraction, therefore  $W = \lambda x. s'$ , **pred**, **succ**, **iszero**, **Y**, **cond**, **cond p** or **cond p q**.

- If  $W = \lambda x. s'$ , we have two cases:

- $x \in \text{fv}(s')$ : then  $\langle W \rangle = \lambda x. [x] \langle s' \rangle =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , thus  $[x] \langle s' \rangle =_{\mathcal{L}_{\text{rec}}} s$ . Since  $s[\langle v \rangle/x] \Downarrow V$  and  $s[\langle v \rangle/x] =_{\mathcal{L}_{\text{rec}}} [x] \langle s' \rangle [\langle v \rangle/x]$  then, by Lemma 4.2  $[x] \langle s' \rangle [\langle v \rangle/x] \rightarrow^* \langle s' \rangle [\langle v \rangle/x]$ , which, by Lemma 4.1, equals  $\langle s'[v/x] \rangle$ , therefore (by Lemma 6)  $\langle s'[v/x] \rangle \Downarrow V''$ , and  $V =_{\mathcal{L}_{\text{rec}}} V''$ . By I.H.,  $s'[v/x] \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle = V$ , therefore  $uv \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V'' =_{\mathcal{L}_{\text{rec}}} V$ .
- $x \notin \text{fv}(s')$ : let  $v'$  represent the term  $\lambda y. \mathcal{E}(\mathcal{E}(y, \langle B \rangle) \multimap \langle B \rangle) x, \langle A \rangle$ . Then  $\langle W \rangle = \lambda x. (\text{rec } \langle 0, 0 \rangle I v' I) \langle s' \rangle =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , therefore  $(\text{rec } \langle 0, 0 \rangle I v' I) \langle s' \rangle =_{\mathcal{L}_{\text{rec}}} s$ . Note that  $s[\langle v \rangle/x] = (\text{rec } \langle 0, 0 \rangle I v' I [\langle v \rangle/x] I) \langle s' \rangle$  and  $(\text{rec } \langle 0, 0 \rangle I v' I [\langle v \rangle/x] I) \langle s' \rangle \Downarrow V$  if  $\langle s' \rangle \Downarrow V$ , then, since  $s'[v/x] = s'$ , by I.H.,  $s' \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$ , therefore  $uv \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$  as required.

- $W = \text{succ}$ : then  $\langle W \rangle = \lambda x. \text{rec } \langle x, 0 \rangle S 0 (\lambda x. Sx) I =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , then  $\text{rec } \langle x, 0 \rangle S 1 (\lambda x. Sx) I =_{\mathcal{L}_{\text{rec}}} s$ . Then  $s[\langle v \rangle/x] = \text{rec } \langle \langle v \rangle, 0 \rangle S 0 (\lambda x. Sx) I$  and  $s[\langle v \rangle/x] \Downarrow V$  if  $\langle v \rangle \Downarrow W'$ , in which case we have two possibilities:

- $W' = 0$ : then  $\text{rec } \langle \langle v \rangle, 0 \rangle \text{ S } 0 (\lambda x. Sx) I \Downarrow V$  if  $\text{S } 0 \Downarrow V$ , in which case  $V = \text{S } 0$ . By I.H.,  $v \Downarrow_{\text{PCF}} W''$ , and  $\langle W'' \rangle =_{\mathcal{L}_{\text{rec}}} 0$ , therefore  $W'' = 0$  (0 is the only value of type  $\mathbf{N}$  that compiles to 0). Therefore  $\text{succ } v \Downarrow_{\text{PCF}} 1$  and  $\langle 1 \rangle = \text{S } 0 =_{\mathcal{L}_{\text{rec}}} V$ .
  - $W' = \text{Sp}$ : then  $\text{rec } \langle \langle v \rangle, 0 \rangle \text{ S } 0 (\lambda x. Sx) I \Downarrow V$  if  $(\lambda x. Sx)(\text{rec } \langle p, 0 \rangle \text{ S } 0 (\lambda x. Sx) I) \Downarrow V$ . By I.H.,  $v \Downarrow_{\text{PCF}} W''$ , and  $\langle W'' \rangle =_{\mathcal{L}_{\text{rec}}} \text{Sp}$ , thus  $W'' = n+1$  ( $W''$  is a number in PCF and it must be different from 0, otherwise its compilation would be 0) and  $p =_{\mathcal{L}_{\text{rec}}} \text{S}^n 0$ . Note that  $(\lambda x. Sx)(\text{rec } \langle \text{S}^n 0, 0 \rangle \text{ S } 0 (\lambda x. Sx) I) \rightarrow^* \text{S}^{n+2} 0$ , therefore, by Lemma 6,  $V =_{\mathcal{L}_{\text{rec}}} \text{S}^{n+2} 0$ . Now it suffices to notice that  $\text{succ } v \Downarrow_{\text{PCF}} n+2$ , and  $\langle n+2 \rangle = \text{S}^{n+2} 0 =_{\mathcal{L}_{\text{rec}}} V$  as required.
- For `pred` and `iszero`, the proof is similar to the case of `succ`.
  - If  $W = Y_A$ : let  $w'$  represent the term  $(\lambda y. \text{let } \langle y_1, y_2 \rangle = y \text{ in } \langle \text{S}(y_1), y_2 \rangle)$ . Then  $\langle W \rangle = \lambda x. \text{rec } \langle \text{S}(0), 0 \rangle \mathcal{M}(\langle A \rangle) x w' =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , therefore  $\text{rec } \langle \text{S}(0), 0 \rangle \mathcal{M}(\langle A \rangle) x w' =_{\mathcal{L}_{\text{rec}}} s$ . Then, since  $s[\langle v \rangle/x] = \text{rec } \langle \text{S}(0), 0 \rangle \mathcal{M}(\langle A \rangle) \langle v \rangle w'$ ,  $s[\langle v \rangle/x] \Downarrow V$  if  $\langle v \rangle(\langle Y_A \rangle \langle v \rangle) \Downarrow V$  (and  $\langle v \rangle(\langle Y_A \rangle \langle v \rangle) = \langle v(Y_A v) \rangle$ ). Thus, by I.H.  $v(Y_A v) \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$ , therefore  $Y_A v \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$  as required.
  - $W = \text{cond}_A$ : let  $v'$  represent the term  $(\lambda z. (\text{rec } \langle 0, 0 \rangle I \mathcal{E}(z, \langle A \rangle) I) q)$ . Then  $\langle W \rangle = \lambda x p q. \text{rec } \langle x, 0 \rangle p v' I =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , therefore  $\lambda p q. \text{rec } \langle x, 0 \rangle p v' I =_{\mathcal{L}_{\text{rec}}} s$ . Then  $s[\langle v \rangle/x] = \lambda p q. \text{rec } \langle \langle v \rangle, 0 \rangle p v' I$  and  $s[\langle v \rangle/x] \Downarrow \lambda p q. \text{rec } \langle \langle v \rangle, 0 \rangle p v' I$ . Note that  $\text{cond}_A v \Downarrow_{\text{PCF}} \text{cond}_A v$ , because it is a value, and  $\langle \text{cond}_A v \rangle =_{\mathcal{L}_{\text{rec}}} \lambda p q. \text{rec } \langle \langle v \rangle, 0 \rangle p v' I$ .
  - $W = \text{cond}_A p_1$ : let  $v'$  represent the term  $(\lambda z. (\text{rec } \langle 0, 0 \rangle I \mathcal{E}(z, \langle A \rangle) I) q)$ . Then  $\langle W \rangle = (\lambda x p q. \text{rec } \langle x, 0 \rangle p v' I) \langle p_1 \rangle =_{\mathcal{L}_{\text{rec}}} \lambda p q. \text{rec } \langle \langle p_1 \rangle, 0 \rangle p v' I =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , therefore  $\lambda p q. \text{rec } \langle \langle p_1 \rangle, 0 \rangle p v' I =_{\mathcal{L}_{\text{rec}}} s$ . Then  $s[\langle v \rangle/x] = \lambda q. \text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle v \rangle v' I$  and  $s[\langle v \rangle/x] \Downarrow \lambda q. \text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle v \rangle v' I$ . Note that  $\text{cond}_A p_1 v \Downarrow_{\text{PCF}} \text{cond}_A p_1 v$ , because it is a value, and  $\langle \text{cond}_A p_1 v \rangle =_{\mathcal{L}_{\text{rec}}} \lambda y. \text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle v \rangle v' I$ .
  - $W = \text{cond}_A p_1 p_2$ : let  $v'$  represent the term  $(\lambda z. (\text{rec } \langle 0, 0 \rangle I \mathcal{E}(z, \langle A \rangle) I) x)$ . Then  $\langle W \rangle = \lambda x. \text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle p_2 \rangle v' I =_{\mathcal{L}_{\text{rec}}} \lambda x. s$ , therefore  $\text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle p_2 \rangle v' I =_{\mathcal{L}_{\text{rec}}} s$ . Then  $s[\langle v \rangle/x] = \text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle p_2 \rangle v' [\langle v \rangle/x] I$  and  $s[\langle v \rangle/x] \Downarrow V$  if  $\langle p_1 \rangle \Downarrow W'$ , in which case we have two possibilities:
    - $W' = 0$ : then  $\text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle p_2 \rangle v' [\langle v \rangle/x] I \Downarrow V$  if  $\langle p_2 \rangle \Downarrow V$ . By I.H.,  $p_1 \Downarrow_{\text{PCF}} W''$ , and  $\langle W'' \rangle =_{\mathcal{L}_{\text{rec}}} 0$ , therefore  $W'' = 0$  (0 is the only value of type  $\mathbf{N}$  that compiles to 0). Also by I.H.,  $p_2 \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$ , therefore  $\text{cond}_A p_1 p_2 v \Downarrow_{\text{PCF}} V'$ , thus  $uv \Downarrow_{\text{PCF}} V'$ , and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$  as required.
    - $W' = \text{Sp}'$ : then  $\text{rec } \langle \langle p_1 \rangle, 0 \rangle \langle p_2 \rangle v' [\langle v \rangle/x] I \Downarrow V$  if  $\langle v \rangle \Downarrow V$ . By I.H.,  $p_1 \Downarrow_{\text{PCF}} W''$ , and  $\langle W'' \rangle =_{\mathcal{L}_{\text{rec}}} \text{Sp}'$ , thus  $W'' = n+1$  ( $W''$  is a number in PCF and it must be different from 0, otherwise its compilation would be 0). Also by I.H.,  $t \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$ , therefore  $\text{cond}_A p_1 p_2 v \Downarrow_{\text{PCF}} V'$  and  $\langle V' \rangle =_{\mathcal{L}_{\text{rec}}} V$  as required.

□

This completes the proof of soundness and completeness of the encoding.

Note that the terms of the form  $\text{rec } \langle 0, 0 \rangle I t I$  used in the encoding of conditionals and  $\lambda$ -abstractions allow us to discard terms without evaluating them. This is a feature of the encoding, otherwise terminating programs in PCF could be translated to non-terminating programs in System  $\mathcal{L}_{\text{rec}}$ . This differs from the definition of erasing given in Section 3, where terms are consumed and not discarded (in pure linear systems functions do not discard their arguments). However, allowing terms to be discarded without being evaluated, is crucial when defining recursion based on fixpoints.

Once a PCF term is compiled into  $\mathcal{L}_{\text{rec}}$  it can be implemented using the techniques in Section 4, thus we obtain a new stack machine implementation of PCF.

## 6 Conclusions

Our work previously investigated linear primitive recursive functions, and a linear version of Gödel's System  $\mathcal{T}$ . This paper shows how the essential ideas can be extended to general recursion, as found in programming languages.  $\mathcal{L}_{\text{rec}}$  is a syntactically linear calculus, but only the fragment without the recursor is operationally linear. The linear recursor allows us to encode duplicating and erasing, thus playing a similar role to the exponentials in linear logic. It encompasses bounded recursion (iteration) and minimisation in just one operator, thus  $\mathcal{L}_{\text{rec}}$  can be seen as an alternative way to recover the power of the  $\lambda$ -calculus within a linear system. The meaning of this linear recursor will be further analysed in a denotational setting in future work.

The pragmatismal impact of these results is currently being investigated within the language Lilac [49]. Other work that is currently being investigated includes the relationship with calculi for explicit computational complexity. For instance, closed construction in System  $\mathcal{L}$  (that is, the requirement that in any iterator term the function to be iterated must be closed when the term is constructed, as opposed to being closed at the moment of reduction) gives PR functions [18], but closed construction does not affect System  $\mathcal{L}_{\text{rec}}$  (the encoding of  $\mu$  is closed).

## References

- [1] S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.
- [2] S. Alves. *Linearisation of the Lambda Calculus*. PhD thesis, Faculty of Science - University of Porto, April 2007.
- [3] S. Alves, M. Fernández, M. Florido, and I. Mackie. Linear recursive functions. In *Rewriting, Computation and Proof*, pages 182–195, 2007.
- [4] S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's system  $\mathcal{T}$  revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010.
- [5] S. Alves, M. Fernández, M. Florido, and I. Mackie. Recursion in linear typed lambda-calculus, 2011. Submitted.
- [6] S. Alves and M. Florido. Weak linearization of the lambda calculus. *Theoretical Computer Science*, 342(1):79–103, 2005.
- [7] A. Asperti. Light affine logic. In *Proc. Logic in Computer Science (LICS'98)*, pages 300–308. IEEE Computer Society, 1998.
- [8] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM Transactions on Computational Logic*, 3(1):137–175, 2002.
- [9] P. Baillot and V. Mogbil. Soft lambda-calculus: a language for polynomial time computation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*, pages 27–41. Springer-Verlag, 2004.
- [10] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
- [11] U. Berger. Minimisation vs. recursion on the partial continuous functionals. In *In the Scope of Logic, Methodology and Philosophy of Science*, volume 1 of *Synthese Library 316*, pages 57–64. Kluwer, 2002.
- [12] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science, 15-18 July, 1991, Amsterdam, The Netherlands*, pages 203–211. IEEE Computer Society, 1991.

- [13] G. M. Bierman, A. M. Pitts, and C. V. Russo. Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Fourth International Workshop on Higher Order Operational Techniques in Semantics, Montréal*, volume 41 of *ENTCS*, pages 70–88. Elsevier, Sept. 2000.
- [14] G. Boudol. The lambda-calculus with multiplicities (abstract). In *CONCUR'93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 1993.
- [15] G. Boudol, P.-L. Curien, and C. Lavatelli. A semantics for lambda calculi with resources. *Mathematical Structures in Computer Science*, 9(4):437–482, 1999.
- [16] T. Braüner. The Girard translation extended with recursion. In *Computer Science Logic, 8th International Workshop, CSL'94, Kazimierz, Poland*, volume 933 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 1994.
- [17] P.-L. Curien. An abstract framework for environment machines. *Theor. Comput. Sci.*, 82(2):389–402, 1991.
- [18] U. Dal Lago. The geometry of linear higher-order recursion. In *IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 366–375, June 2005.
- [19] V. de Paiva and E. Ritter. Variations on Linear PCF, 1999. Proceedings of WESTAPP-99.
- [20] N. Dershowitz. Term rewriting systems by “terese”, cambridge tracts in theoretical computer science 55. *Theory Pract. Log. Program.*, 5(3):395–399, 2005.
- [21] M. C. J. D. v. E. E. G. J. M. H. Nöcker, J. E. W. Smetsers and M. J. Plasmeijer. Concurrent clean. In *PARLE'91 Parallel Architectures and Languages Europe*, volume 506 of *LNCS*, pages 202–219. Springer, 1991.
- [22] T. Ehrhard and L. Regnier. The differential lambda-calculus. *Theoretical Computer Science*, 309(1-3):1–41, 2003.
- [23] T. Ehrhard and L. Regnier. Uniformity and the taylor expansion of ordinary lambda-terms. *Theoretical Computer Science*, 403(2-3):347–372, 2008.
- [24] M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *MSCS*, 15(2):343–381, 2005.
- [25] M. Fernández, I. Mackie, and F. R. Sinot. Lambda-calculus with director strings. *Applicable Algebra in Engineering, Communication and Computing*, 15(6):393–437, 2005.
- [26] M. Fernández and N. Siafakas. New developments in environment machines. *Electr. Notes Theor. Comput. Sci.*, 237:57–73, 2009.
- [27] M. Gaboardi and L. Paolini. Syntactical, operational and denotational linearity. In *Workshop on Linear Logic, Ludics, Implicit Complexity and Operator Algebras. Dedicated to Jean-Yves Girard on his 60th birthday*, Siena, May 2007.
- [28] D. R. Ghica. Geometry of synthesis: a structured approach to vlsi design. In *POPL*, pages 363–375, 2007.
- [29] J. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.
- [30] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
- [31] J.-Y. Girard. Towards a geometry of interaction. In *Categories in Computer Science and Logic: Proc. of the Joint Summer Research Conference*, pages 69–108. American Mathematical Society, 1989.

- [32] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
- [33] J.-Y. Girard, A. Scedrov, and P. J. Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992.
- [34] M. Giunti and V. T. Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR*, pages 432–446, 2010.
- [35] C. Hankin. *An Introduction to Lambda Calculi for Computer Scientists*, volume 2. College Publications, 2004. ISBN 0-9543006-5-3.
- [36] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proc. Logic in Computer Science (LICS'99)*. IEEE Computer Society, 1999.
- [37] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.
- [38] S. Holmström. Linear functional programming. In *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, 1988.
- [39] K. Honda. Types for dyadic interaction. In E. Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer Berlin / Heidelberg, 1993.
- [40] A. J. Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000.
- [41] S. C. Kleene. *Introduction to Metamathematics*. North-Holland, 1952.
- [42] J. W. Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematisch Centrum, Amsterdam, 1980.
- [43] J. W. Klop. New fixpoint combinators from old. *Reflections on Type Theory*, 2007.
- [44] J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theor. Computer Science*, 121:279–308, 1993.
- [45] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.
- [46] N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *POPL*, pages 358–371, 1996.
- [47] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988.
- [48] Y. Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1-2):163–180, 2004.
- [49] I. Mackie. Lilac: A functional programming language based on linear logic. *Journal of Functional Programming*, 4(4):395–433, 1994.
- [50] I. Mackie. The geometry of interaction machine. In *Principles of Programming Languages (POPL)*, pages 198–208. ACM Press, 1995.
- [51] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1 – 40, 1992.
- [52] L. Paolini and M. Piccolo. Semantically linear programming languages. In *PPDP*, pages 97–107, Valencia, Spain, 2008. ACM.
- [53] G. D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

- [54] K. Terui. Light affine calculus and polytime strong normalization. In *Proc. Logic in Computer Science (LICS'01)*. IEEE Computer Society, 2001.
- [55] A. v. Tonder. A lambda calculus for quantum computation. *SIAM J. Comput.*, 33(5):1109–1135, 2004.
- [56] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.
- [57] D. Walker. Substructural type systems. In *Advanced Topics in Types and Programming Languages*, chapter 1, pages 3–43. MIT Press, Cambridge, 2005.
- [58] K. Wansbrough and S. P. Jones. Simple usage polymorphism. In *Proceedings of the Third ACM SIGPLAN Workshop on Types in Compilation, Montreal, Canada*. ACM Press, 2000.
- [59] N. Yoshida, K. Honda, and M. Berger. Linearity and bisimulation. In *FoSSaCS*, pages 417–434, 2002.